UC Santa Cruz

UC Santa Cruz Electronic Theses and Dissertations

Title

Understanding and Simulating Software Evolution

Permalink

<https://escholarship.org/uc/item/4qx5254b>

Author

Lin, Zhongpeng

Publication Date

2015

Copyright Information

This work is made available under the terms of a Creative Commons Attribution License, available at <https://creativecommons.org/licenses/by/4.0/>

Peer reviewed|Thesis/dissertation

UNIVERSITY OF CALIFORNIA
SANTA CRUZ

UNDERSTANDING AND SIMULATING SOFTWARE EVOLUTION

A dissertation submitted in partial satisfaction of the
requirements for the degree of

DOCTOR OF PHILOSOPHY

in

COMPUTER SCIENCE

by

Zhongpeng Lin

December 2015

The Dissertation of Zhongpeng Lin
is approved:

Prof. Jim Whitehead, Chair

Asst. Prof. Seshadhri Comandur

Prof. Tim Menzies

Tyrus Miller
Vice Provost and Dean of Graduate Studies

Copyright © by

Zhongpeng Lin

2015

Table of Contents

List of Figures	v
List of Tables	vii
Abstract	ix
Dedication	xi
Acknowledgments	xii
1 Introduction	1
1.1 Emergent Phenomena in Software	1
1.2 Simulation of Software Evolution	3
1.3 Research Outline	4
2 Power Law and Complex Networks	6
2.1 Power Law	6
2.2 Complex Networks	9
2.3 Empirical Studies of Software Evolution	12
2.4 Summary	17
3 Data Set and AST Differences	19
3.1 Data Set	19
3.2 ChangeDistiller	21
3.3 Data Collection Work Flow	23
4 Change Size in Four Open Source Software Projects	24
4.1 Methodology	25
4.2 Commit Size	27
4.3 Monthly Change Size	32
4.4 Summary	36

5	Generative Models for Power Law and Complex Networks	38
5.1	Generative Models for Power Law	38
5.1.1	Preferential Attachment	41
5.1.2	Self-organized Criticality	42
5.2	Generative Models for Complex Networks	50
6	Simulating SOC and Preferential Attachment in Software Evolution	53
6.1	Preferential Attachment	54
6.2	Self-organized Criticality	56
6.3	Simulation Model	57
6.4	Experiment Setup	59
6.5	Simulation Results	60
6.5.1	Commit Size	61
6.5.2	Distributions of Static Code Measures	64
6.5.3	Other Graph Measures	64
6.6	Summary and Discussion	66
7	Agent-based Model for Software Evolution	69
7.1	Simulation Model	70
7.2	Experiment Setup	73
7.3	Simulation Results	74
7.3.1	Commit Size	74
7.3.2	Developers	76
7.3.3	Distributions of Static Code Measures	77
7.3.4	Other Graph Measures	80
7.4	Summary and Discussion	80
8	Related Models for Software Evolution	82
9	Future Work	86
9.1	Extension of the Models	86
9.2	Empirical Verification of the Generative Mechanisms	87
10	Discussion and Conclusion	89

List of Figures

1.1	A durability simulator (Location: IKEA, East Palo Alto, California). .	4
2.1	Average clustering coefficient $C(k)$ of all k -degree vertices in class collaboration graphs of six software systems: (a) VTK, (b) Digital Material, (c) AbiWord, (d) Linux kernel, (e) MySQL, (f) XMMS (figures reprinted from Myers, 2003)	14
3.1	A simple Git commit graph. Black nodes are commits in the “master” branch; gray nodes are not.	21
4.1	CCDF for commit size distribution.	29
4.2	CCDF for monthly change size distribution.	34
5.1	Wildfire started by lightning that burns an entire cluster (reprinted from Newman, 2005).	46
5.2	The lowest fitness value over time (reprinted from Paczuski et al., 1996). The x-axis is the time, and the dots are the fitness of the least fit species at the time. The step function represents the highest fitness number of the least fit species ever appear before the time. . .	48
6.1	Minimal fitness value at different steps. The x-axis represents different steps, y-axis is the minimal fitness value of all methods at the end of each step.	61
6.2	The CCDFs of commit size distributions during the simulations . . .	62
6.3	The CCDFs of method fan-in and class size distributions after simulations Run #1	65
6.4	Properties of class graph in simulation Run #1	65
7.1	The CCDFs of commit size distributions during the simulations . . .	76
7.2	The number of developers over time in Run #20	77
7.3	The CCDFs of method fan-in and class size distributions after simulations Run #20	78

7.4	Properties of class graph in simulation Run #20	78
7.5	The CCDFs of method fan-in, class size and collaborators distributions after simulations Run #7	79

List of Tables

3.1	Data source	20
4.1	Order statistics for commit size	28
4.2	Parameters and p -values of fitting power law distributions to commit size distributions	29
4.3	Comparing power law with alternative distributions for goodness of fit to commit size data	31
4.4	Parameters and p -values of fitting log-normal distributions to commit size distributions	32
4.5	Order statistics for monthly change size	33
4.6	Parameters and p -values of fitting power law distributions to monthly change size distributions	33
4.7	Comparing power law with alternative distributions for goodness of fit to monthly change size data	33
4.8	Parameters and p -values of fitting log-normal distributions to monthly change size distributions	36
5.1	Hurst exponent of change size in different projects	49
6.1	Fitting power law to commit size distributions in Run #1 and #3 . . .	62
6.2	p , H values and graph measures of different simulation runs. The p -values are from the goodness-of-fit tests of power law to commit size distributions (p_1), method fan-in distributions (p_2), class size distributions (p_3), and class collaborators distributions (p_4). p values less than 0.05 are shown in bold. The H values are the Hurst exponents of the commit sizes. The graph measures are: characteristic path length (d) of the largest connected component, clustering coefficient of the entire graph (C) and the random graph of the same size (C_{random})	63
6.3	Fitting power law to in-degree distributions of method call graphs in Run #1 and #3	65
6.4	Fitting power law to class size distributions in Run #1 and #20 . . .	66

6.5	Fitting power law to degree distributions of class graphs in Run #1 and #10	66
7.1	p , H values and graph measures of different simulation runs. The p -values are from the goodness-of-fit tests of power law to commit size distributions (p_1), method fan-in distributions (p_2), class size distributions (p_3), and class collaborators distributions (p_4). p values less than 0.05 are shown in bold. The graph measures are: characteristic path length (d) of the largest connected component, clustering coefficient of the entire graph (C) and the random graph of the same size (C_{random})	75
7.2	Fitting power law to commit size distributions in Run #20 and #18 .	75
7.3	Fitting power law to in-degree distributions of method call graphs in Run #20 and #7	78
7.4	Fitting power law to class size distributions in Run #20 and #7 . . .	79
7.5	Fitting power law to degree distributions of class graphs in Run #20 and #7	79

Abstract

Understanding and Simulating Software Evolution

by

Zhongpeng Lin

The important roles of software in modern life have prompted people to study software evolution. This dissertation takes a scientific perspective to model the dynamics of software evolution, and verify the model by simulation.

Empirical studies of software evolution have discovered many emergent phenomena, such as the power law distributions in different metrics of software systems and the complex network of relationships among software components. By reviewing the literature, we found similar phenomena in other fields too, as well as generative mechanisms proposed to account for the phenomena. A model of software evolution that incorporates preferential attachment and self-organized criticality (SOC) is proposed in this thesis research and the simulation runs using this model reproduce several emergent phenomena in software evolution. To further explore how individual developers' local changes accumulate to produce high-level phenomena, and to explore the possible causes of log-normal behavior at the body of the commit size distribution, we propose another model based on agents to simulate the interaction between developers and a manager. This simulation shows that power law distributions can emerge from low-level changes without explicit preferential attachment process, and the lower-end of commit size distribution may be

shaped by log-normal task size.

This dissertation research reveals that large software changes are inevitable in software evolution, and the change size follows a heavy-tailed distribution, making it very difficult to estimate. The simulations also show that large changes are not necessarily caused by significant external factors; the power law structure of software systems can also lead to large changes, as part of the power law change size.

To my wife, Yuting Gao, for her patience

To my parents, for their support

Acknowledgments

The text of this dissertation includes reprints of the following previously published material:

- Lin, Z. and Whitehead, J. (2015). Why Power Laws? An Explanation from Fine-Grained Code Changes. In *Proceedings of the 12th Working Conference on Mining Software Repositories*, pages 68–75.
- Lin, Z. and Whitehead, J. (2014). Using fine-grained code change metrics to simulate software evolution. In *Proceedings of the 5th International Workshop on Emerging Trends in Software Metrics*, pages 15–18.
- Lin, Z. (2013). Understanding and simulating software evolution. In *Proceedings of the 35th International Conference on Software Engineering*, pages 1411–1414.

The co-author listed in these publications directed and supervised the research which forms the basis for the dissertation.

Chapter 1

Introduction

1.1 Emergent Phenomena in Software

Software systems are fully intentional products of human intelligence. Hence, one should expect that every aspect of software, except for defects due to human mistakes, will encode some human intention: either from a black box point of view, such as the behavior of an interface element, or from a white box point of view, such as the way a method calls another method.

Nevertheless, empirical studies of software systems have revealed phenomena beyond human intention. A well-known example is the prevalence of power laws, found in the distributions of file size (Herraiz et al., 2011), file change size (Gorshenev and Pis'mak, 2004), commit size (Hattori and Lanza, 2008), number of subclasses, variable and method names (Turnu et al., 2011), and many more.

Computer software often consists of many components of different granu-

larity, such as libraries, modules, classes, and functions. These components are not isolated, but instead have various inter-relationships, via function/method calls, class inheritance, package imports, etc. Hence, software can be modeled as a network, with the components as vertices and their relations as the edges.

Looking into software component networks, it has been found that they are not random graphs, but instead exhibit properties of complex networks. Class graphs, method call graphs, and library and package dependency networks in software systems are all *scale-free networks*, i.e., their degree distributions follow a power law (Valverde and Solé, 2003; Louridas et al., 2008; Turnu et al., 2011). Meanwhile, Valverde and Solé (2003) also found that software class graphs have high degree of clustering—another feature not found in random graphs.

Given the multiplicity of empirical studies across many different kinds of software systems, the power law distributions and the properties of complex networks are unlikely to be accidental. Some mechanism is at work to create these properties, but which one? Developers might naturally come to mind, since it is their hands that write the code that exhibits these phenomena. Yet, assuredly, no developer intentionally sets out to create them. We have never heard a developer say, “the distribution of file sizes on this project is diverging from power law, so we should make changes to fix this!” Indeed, accomplishing such a policy would involve known bad practices. For example, power law tails in file size distributions indicate the existence of large files, a code smell (Fowler and Beck, 1999). Nevertheless, this code smell exists even in projects believed to be well-maintained,

such as those studied by Herraiz et al. (2011). Rather than the deliberate result of intentional activity, power laws and the properties of complex networks must be emergent phenomena caused by software evolution dynamics not yet fully understood.

1.2 Simulation of Software Evolution

It is challenging to explore the dynamics leading to the emergent phenomena in software evolution, as they are caused by the cumulative effect of a software change process working over time. Achieving a high degree of control over a real world change process is challenging, and expensive. There is no cost-effective way to set up a controlled multi-year software evolution experiment in a realistic setting. Instead, we use software evolution simulations as a way to focus on interesting factors within a complex overall process, and study possible causes of power law distributions and complex networks.

Simulations are widely used in many areas to model natural systems, human systems or technologies for various purposes.¹ They are often used when the real process is not repeatable, e.g., urban development in response to different government policies (Waddell and Ulfarsson, 2004; Deal and Sun, 2006). Simulations can also be effective when the real process takes too long to finish. For example, IKEA uses various simulators to test the durability of furniture (Figure 1.1). Without such simulators, it would take years to observe how a piece of furniture

¹Wikipedia: <https://en.wikipedia.org/wiki/Simulation>

Figure 1.1: A durability simulator (Location: IKEA, East Palo Alto, California).

deteriorates.

A software evolution process is both unrepeatable and longstanding. Simulation has the advantage of permitting a high degree of control over software change processes, and the ability to quickly change parameters and see the impact of these changes over a multi-year evolution.

1.3 Research Outline

Based on empirical studies that revealed emergent power law distributions and complex networks in software evolution, this research builds models to explain the empirical findings, tests the models by simulating them, and compares the outcomes with the empirical findings, thus improving our understanding of the dynamics of software evolution.

The rest of this dissertation starts by reviewing some emergent phenomena found by previous empirical studies (Chapter 2). Not satisfied with the measure of change size in the literature, we propose a new measure and conducted our empirical study of change size based on the new measure (Chapter 3 and 4). Chapter 5 presents generative mechanisms for similar emergent phenomena in other fields, among which self-organized criticality (SOC) and preferential attachment are used in this thesis study. Then Chapter 6 describes a model that uses preferential attachment in a SOC process, yielding distributions and networks resembling actual software evolution processes. To address the issues found in this model, an agent-based model is explored to simulate human interactions during software evolution (Chapter 7). This dissertation then provides an overview of related models for software evolution (Chapter 8), and thoughts about the extension of this work (Chapter 9). Finally, Chapter 10 concludes the dissertation.

Chapter 2

Power Law and Complex Networks

This chapter provides some brief theoretical background on power law distributions and complex networks, as well as empirical evidence from the literature demonstrating their presence in software systems.

2.1 Power Law

Mathematically, a power law distribution can be described by its probability density function:

$$p(x) = Cx^{-\alpha}, \text{ where } C > 0 \text{ and } \alpha > 0$$

The constant α is the *exponent* or *scaling parameter* of the power law, while the constant C is determined by the requirement that the probability function sums up to 1. As this density function diverges when $x \rightarrow 0$, there should be a lower bound to the power law behavior, denoted as x_{min} . In many real world cases,

there is some non-power-law behavior at the lower end of the distribution, making $x_{min} \gg 0$.

Unlike other distributions, a power law distribution does not have a typical or characteristic scale, and the distribution is the same “whatever scale we look at it on” (Newman, 2005). When we increase x by a factor of k , the shape of the distribution does not change. A typical example is the distribution of computer file size. If we find that files of size 2kB are 1/4 as frequent as those of size 1kB, we may also find those of size 2MB to be 1/4 as frequent as those of 1MB, and those of 2GB to be 1/4 as frequent as 1GB, and so on. This is because:

$$\frac{p(kx)}{p(x)} = \frac{C(kx)^{-\alpha}}{Cx^{-\alpha}} = k^{-\alpha}, \forall x > x_{min} \text{ and } k > 1$$

Note that $k^{-\alpha}$ is a constant. Newman (2005) proved that only the power law distribution has this feature. The mean of the power law distribution is given by:

$$E X = \int_{x_{min}}^{\infty} xp(x) dx = C \int_{x_{min}}^{\infty} x^{-\alpha+1} dx = \frac{C}{2-\alpha} x^{-\alpha+2} \Big|_{x_{min}}^{\infty}$$

When $\alpha \leq 2$, the mean of the power law distribution goes to infinity. The infinite mean implies that it is impossible to obtain an accurate estimate of the mean from sample data, because sample data always have a finite average and the average values have very large fluctuations from sample to sample.

When $\alpha > 2$, the mean is given by:

$$E X = \frac{\alpha - 1}{\alpha - 2} x_{min}$$

The second moment of the distribution is given by:

$$E X^2 = \int_{x_{min}}^{\infty} x^2 p(x) dx = C \int_{x_{min}}^{\infty} x^{-\alpha+2} dx = \frac{C}{3-\alpha} x^{-\alpha+3} \Big|_{x_{min}}^{\infty}$$

which diverges when $\alpha \leq 3$. Since the variance of the distribution is given by:

$$\text{Var } X = E X^2 - (E X)^2$$

when $\alpha \leq 3$, the variance diverges as well. When $\alpha > 3$, the variance is well defined, taking the value:

$$\text{Var } X = [\frac{\alpha-1}{\alpha-3} - (\frac{\alpha-1}{\alpha-2})^2] x_{min}^2$$

The above equations and deductions have been focused on power law distributions for continuous random variables. However, many random variables in this thesis study are discrete, such as change size and class size. The difference is trivial in most cases, as only the tail of the distribution follows power law, and the x_{min} is often large enough that the values in the range can be considered continuous. The derivation presented above can be generalized to the discrete case, but it is more tractable to work with integrals.

Visually, a power law distribution can be illustrated using a log-log scale, where its probability function $p(x)$ becomes a straight line:

$$\ln p(x) = \ln(Cx^{-\alpha}) = -\alpha \ln x + \ln C$$

When plotting the frequency distribution of sample data, however, the probability of a large x value is so small that there are very few data points with

large values, resulting in a noisy curve at the right end. As a result, Newman (2005) suggests plotting the complementary cumulative density function (CCDF) instead, which is given by:

$$CCDF(x) = P(X \geq x) = \int_x^\infty C t^{-\alpha} dt = \frac{C}{\alpha - 1} x^{-\alpha+1}, \text{ when } \alpha > 1$$

The CCDF is also a power law function. On the log-log scale, it is a straight line with shallower slope than $p(x)$:

$$\ln P(X \geq x) = (-\alpha + 1) \ln x + \ln \frac{C}{\alpha - 1}$$

2.2 Complex Networks

A complex network is a graph with topological features *not* found in simple networks, such as random graphs. Often used as a benchmark, a random graph with N vertices and M edges is generated by randomly picking M pair of vertices out of all possible vertex pairs to add edges. Hence, the vertex degrees of random graphs follow binomial distributions. Many complex networks, however, have power law degree distributions. Such networks are known as scale-free networks, due to the scale-free nature of the power law.

Some topological features that distinguish complex networks from random graphs are based on the concept of *clustering coefficient*, which measures the tendency of vertices in a graph to cluster together. One can evaluate a graph generation model by comparing the clustering coefficient of the synthesized graph to that of the real graph. It is one of few measures that can be computed efficiently

on very large graphs, and provides useful insights into the overall characteristics of a graph.

Given a vertex v_i with degree k_i , its clustering coefficient is the fraction of vertex pairs in its neighborhood that are linked to each other, and thus forming triangles with v_i . If v_i participates in t_i triangles, the clustering coefficient of v_i is given by:

$$C_i = \frac{2t_i}{k_i(k_i - 1)}$$

A wedge in a graph is defined as a pair of edges with a shared vertex. The global clustering coefficient, also known as *transitivity*,¹ of a graph is the fraction of wedges that form triangles in the graph:

$$C = 3 \times \frac{\#\text{triangles}}{\#\text{wedges}}$$

Take social network for example, if A and B know each other, B and C know each other, clustering coefficient tells us how likely A and C know each other too. In random graphs, whether there is a edge between two vertices is independent of the vertices, so the clustering coefficient of a random graph is simply the probability of any pair of vertices being connected, which is its density:

$$C_{\text{random}} = D = \frac{2M}{N(N - 1)} \quad (2.1)$$

The clustering coefficient of complex networks is often much higher than that of random graphs with the same size (i.e., same number of vertices and edges). For example, in the collaboration graph of film actors, its clustering coefficient is

¹Clustering coefficient: https://en.wikipedia.org/wiki/Clustering_coefficient

almost 3000 times greater than the corresponding random graph's (Watts et al., 1998).

One can also group vertices by their degrees, calculate the average clustering coefficient for each group, and see how degrees of vertices affect their clustering coefficient. In random graphs, the clustering coefficient of a vertex does not depend on the degree of the vertex, so the average clustering coefficient of all k -degree vertices is approximately the density of the graph:

$$C(k) \approx D = \frac{2M}{N(N-1)}$$

In many complex networks, however, the clustering coefficient is negatively correlated to degree, following the scaling law:

$$C(k) \propto k^{-1}$$

Another useful metric is the characteristic path length of a graph, which measures how tightly connected a graph is. The path length between two vertices in a graph is the number of edges in the shortest path between them. One can calculate path length between each pair of vertices in a graph and take the average over all vertex pairs in the graph, which is the characteristic path length of the graph. Short characteristic path length means that it typically takes only a few edges to travel from any vertex to almost any other vertex in the graph. In some graphs, there is not always a path between any pair of vertices, i.e., the graphs are not connected. In this case, one can only measure the characteristic path length for each connected component—a subgraph in which there is at least a path between

any pair of vertices. A connected graph itself is a connected component. The characteristic path length is often very short in large real world networks. A famous example is the social network of the world. A well-accepted theory by Milgram (1967), known as “six degrees of separation”, states that many pairs of seemingly distant people can be connected by a short acquaintance chain with a typical length of six.

If a network has high clustering coefficient and short characteristic path length, it is called a *small world* by Watts et al. (1998)—a definition of small world that is commonly used in previous empirical studies of software systems (e.g., Valverde et al., 2002; de Moura et al., 2003), and will be followed throughout this dissertation.

2.3 Empirical Studies of Software Evolution

Software systems are complex systems with many interacting components. It is natural to model them as graphs. While software engineering practitioners and researchers have used such graphs to express software architecture and design, identify potential design issues, etc., some other researchers, especially physicists, have been intrigued by the overall properties of these graphs.

Valverde et al. (2002) pioneered this stream of studies by looking at the class diagram of the Java Development Kit (JDK), thereby discovering that it has a scale-free and small-world structure. Wheeldon and Counsell (2003) built separate

graphs for different types of couplings between Java classes: inheritance, interface, aggregation, parameter type and return type. They found power law in the degree distributions of all five types of graphs in JDK, Apache Ant and Tomcat projects.

Myers (2003) studied the class collaboration graphs for three C++ systems and call graphs for three procedural systems written in C. In the class collaboration graphs, each vertex is a class, and a directed edge goes from class A to class B when A makes a reference to B through inheritance or aggregation. In the call graphs, vertices are subroutines, and edges go from callers to callees. Myers found power law behavior in the distributions of both in-degrees and out-degrees. The class collaboration graphs and call graphs can also be converted to undirected graphs by removing the directions on the edges. In the undirected class collaboration graphs, a k degree vertex means that the class has references to or from k other different classes, i.e., it has k neighbors; while a k degree vertex in call graphs means that the subroutine calls or is called by k other different subroutines. With k neighbors, there are $k(k-1)/2$ pairs of classes or subroutines. The clustering coefficient of the vertex is defined as the portion of these pairs that also have references or calls to each other. Clustering coefficient of a class or a subroutine measures the coupling of software components at its neighborhood. Myers found that the more neighbors a class or subroutine has, the less coupled its neighbors are on average, as indicated by the average clustering coefficient of k -degree vertices drops as k increases (Figure 2.1).

For C/C++ systems, graphs can also be built from their header files. De

Figure 2.1: Average clustering coefficient $C(k)$ of all k -degree vertices in class collaboration graphs of six software systems: (a) VTK, (b) Digital Material, (c) Abi-Word, (d) Linux kernel, (e) MySQL, (f) XMMS (figures reprinted from Myers, 2003)

Moura et al. (2003) explored such graphs where two header files are considered functionally related if they are included in the same source file. The graphs of header files in Linux kernel, XFree86 and Mozilla were found to be scale-free, with small-world structure.

Using a larger data set of 80 software systems in C++ and Java, Valverde and Solé (2003) examined the class graphs of inheritance and membership, and again found scale-free and small-world properties. Similar empirical studies in other types of software systems found power laws in SmallTalk class graph (Concas et al., 2007), dependency graphs of Debian packages (Maillart et al., 2008), Perl packages, shared libraries in Unix and Linux distributions, Windows Dynamically Linked Libraries, FreeBSD ports, T_EX and Metafont modules, Ruby libraries, and others (Louridas et al., 2008).

Power law degree distributions were not only observed in static graphs, but also in runtime object graphs. Taking 60 runtime snapshots from 35 Java programs, Potanin et al. (2005) found power law distributions in both in-degree and out-degree of the object graphs. Later, other properties of software graphs were also explored, such as motif (Valverde and Solé, 2005), community structure (Šubelj and Bajec, 2011) and PageRank (Bhattacharya et al., 2012).

Besides degree distributions of software graphs, power law was also found in source code size, such as Java method and class size (Concas et al., 2007) and C/C++ file size (Herraiz et al., 2011).

Change size is another area where power law distributions were found.

Gorshenev and Pis'mak (2004) studied the CVS repositories of the Mozilla web browser, FreeBSD and GNU Emacs, and found power law behaviors in their commit sizes, measured by the number of added and deleted lines. Also from CVS repositories, Wu et al. (2007) found the logical and structural change sizes of nine C/C++ software systems followed power law distribution. In their study, logical change size is defined as the commit size as measured by the number of files changed, while structure changes are defined as the differences in the file dependency graphs obtained by comparing daily snapshots. Hattori and Lanza (2008) confirmed that commit size, measured by the number of changed files, closely follows power laws in nine software projects developed in many different programming languages. Using a very large dataset of 9363 projects, Arafat and Riehle (2009) confirmed similar commit size distributions again, with commit size measured by changed lines of code.

Power law also exists in many other measures of software systems, such as the number of methods, fields, constructors, subclasses, instance variable names and method names (Wheeldon and Counsell, 2003; Concas et al., 2007; Turnu et al., 2011), as well as the number of bugs in a module (Concas et al., 2011).

However, not all measures follow power law. It was found that the out-degree of dependency networks in some systems do not exhibit power law behavior (Concas et al., 2007; Kohring, 2009). Baxter et al. (2006) studied 17 measures on a corpus of 56 Java projects, and found power law in some measures, but not in others. They observed that if the programmers are aware of a property at the time

the software is being written, the property will not follow a power law distribution, as there is a tendency to avoid “big things,” thus truncating the tail of the distribution. For measures such as the number of callers of a method, they are not immediately visible to programmers, so they follow power law distributions.

2.4 Summary

This chapter briefly introduces power law distributions and complex networks, then presents empirical evidence in the literature indicating their existence in software evolution. In our later simulations, we will evaluate our simulation models based on whether they are able to reproduce the empirical findings in software evolution.

As can be seen from Section 2.3, there are different measures for change size in the literature. The easiest way is to count the number of changed files without looking into the file content. However, this is a very coarse measure. In a well-designed software system, changes are often localized. The number of files changed each time may be very small, while the actual change size is large. This problem is avoided when change size is measured at line level by comparing the textual difference between two revisions of source code. Nevertheless, textual line comparison is sensitive to some trivial changes such as formatting. In fact, Hindle et al. (2008) found many large commits were caused by reformatting the code. To overcome the shortcomings of previous work on change size, we define a new

measure for change size and conducted our own empirical study.

In the next two chapters, we first introduce our data collection and extraction approach (Chapter 3). The data analysis methodology and result are presented in Chapter 4.

Chapter 3

Data Set and AST Differences

This chapter describes the data used in this empirical study, as well as the way we extracted the changes from them.

3.1 Data Set

All data come from four open source Java projects: jEdit, Eclipse JDT Core, Apache Maven and Google Guice. CVSanaly¹ was used to collect data from their Git repositories or Git mirrors,² and store them in a MySQL database. We developed the `Content` extension to CVSanaly to store all revisions of all source

¹CVSanaly: <http://metricsgrimoire.github.io/CVSanaly/>

²The Git mirror of jEdit from which we collected data had been removed at the time of writing. For reproducibility, we published our clone of the Git mirror at <https://github.com/linzhp/jEdit-Clone>. The Git repositories of other projects are listed below:

- Eclipse JDT Core: <http://git.eclipse.org/c/jdt/eclipse.jdt.core.git/>
- Apache Maven: <https://github.com/apache/maven.git>
- Google Guice: <https://github.com/google/guice.git>

Table 3.1: Data source

Project	Start Date	End Date	Commits
jEdit	1998-09-27	2012-08-08	6486
Eclipse JDT Core	2001-06-05	2013-09-09	19321
Apache Maven	2003-09-01	2014-01-29	9723
Google Guice	2006-08-22	2013-12-11	1198

code files in a database for easy analysis. Table 3.1 shows the start and end dates of the data collection, as well as the number of commits/revisions collected. Only commits in their “master” branches were used for this empirical study.

A commit of changes to a Git repository produces a new revision. To extract changes in a commit, one has to compare its revision of source code with its previous revision. A naive approach is to sort commits by their time stamp and compare each revision with its preceding one. However, this approach does not work if there is more than one Git branch. As Git keeps the branching and merging history, the revisions in the master branch of a project form a directed acyclic graph, like the black nodes and solid arrows in Figure 3.1. Sorting commits in the “master” branch results in a sequence: A-B-C-D-E-F-H. The naive approach would compare the revision at E with the one at D in order to extract the changes in E. However, as can be seen from Figure 3.1, E is changed from C, rather than D. In order to reliably identify the revision history, we enhanced CVSSanalyzer to maintain a commit graph while loading the repository data.

On the other hand, a commit may be the result of merging two previous revisions due to parallel development, such as F in Figure 3.1. Merge commits

Figure 3.1: A simple Git commit graph. Black nodes are commits in the “master” branch; gray nodes are not.

are ignored in this empirical study, as they normally do not bring meaningful new changes to software other than repeating changes from branches and resolving conflicts. However, some projects examined in this empirical study used other version control systems such as CVS or Subversion at the beginning. After migrating to Git, commits in other branches are often not reachable from the master branch. In this case, merge commits were not excluded from our study, otherwise those branch changes would be lost. Considering Figure 3.1, if the link from D to F is broken, we should not exclude F, even if we know F is a merge commit, for example, from its commit log.

3.2 ChangeDistiller

Source code is semi-structured text; a valid piece of source code can be deterministically parsed into an abstract syntax tree (AST). Code changes can be obtained by comparing the ASTs of two revisions of source code, using tree differ-

encing algorithms. We call the code changes extracted in this manner *AST differences*. The empirical study in this thesis work used ChangeDistiller (Fluri et al., 2007) to extract AST differences. Given two revisions of a piece of Java source code, ChangeDistiller is able to parse each of them into an AST, and use the tree differencing algorithm proposed by Chawathe et al. (1996) to compare the differences between the two ASTs. In the end, ChangeDistiller not only returns the changed nodes, but also classifies the types of changes according to a taxonomy proposed by Fluri and Gall (2006). Each change extracted by ChangeDistiller has information about the type of the changed AST node (e.g., `if` statement, method), the parent and the children of the node, and the change type (e.g., `METHOD_RENAMING`, `ADDITIONAL_CLASS`). It also has information about the position of the node in the source code, allowing for inspection at even finer granularity if necessary.

ChangeDistiller was initially developed as an Eclipse plugin, and later adapted as a stand-alone open source library.³ It is currently only capable of extracting changes in Java code, but it was designed to be extended to support more languages.

ChangeDistiller provides a convenient API for comparing revisions of source code. One drawback, however, is that when a class or method is added or deleted, ChangeDistiller only reports the insertion or deletion of the class or method node,

³The source code of ChangeDistiller can be found at <https://bitbucket.org/sealuzh/tools-changedistiller>. We made several bug fixes and enhancements to ChangeDistiller. However, the authors of the repository did not agree with the implementation of some enhancements. So they accepted our pull requests, hosted those enhancements in a separate branch called “fileversion,” and planned to reimplement the enhancements. Unfortunately, all the authors left the project before the reimplementation. We used the latest revision at the “fileversion” branch of ChangeDistiller in this study.

but not the descendants of the node. For example, when an inner class is added with several methods in it, ChangeDistiller only reports an `ADDITIONAL_CLASS` change, but not the accompanying insertion of methods of the inner class. Fortunately, the changes given by ChangeDistiller have information about the positions (line numbers) of the changed nodes. With the information regarding the start line and end line of an inserted or deleted node, one could parse the source code, and then work out the nodes falling into the line number range, and thereby determining the missing sub-nodes. In this work, the above mentioned approach is used to include the AST nodes within a added or deleted subtree.

3.3 Data Collection Work Flow

To prepare the data for this empirical study, we first ran CVSanaly to load commit data from Git repositories into a MySQL database. Then, we wrote a program⁴ to read every commit in the database, decide which files were changed in each commit, retrieve the revisions before and after the changes for each file, and call ChangeDistiller to extract the AST differences. Finally, the extracted changes were stored in a MongoDB database for further analysis.⁵

⁴The source code of the program can be found at <https://github.com/linzhp/ChangeAnalyzer>.

⁵The MongoDB version we used was 2.6. A dump of the MongoDB is at <https://users.soe.ucsc.edu/~linzhp/dissertation/change-data.tar.gz>.

Chapter 4

Change Size in Four Open Source Software Projects

To avoid the issues of measuring change size discussed in Section 2.4, we count the number of AST differences, and use it as the measure of change size throughout the rest of this dissertation, unless otherwise noted. In this chapter, we analyze the distributions of change size from different angles. Given the outcome of previous studies in change size (see Section 2.3), it is likely that change sizes as measured by the number of AST differences will also follow power law distributions. We will use this as our hypothesis in this section, and perform statistical tests to verify.

4.1 Methodology

A power law distribution is a highly skewed distribution, for which mean and standard deviation give little information. Instead, *order statistics* (Casella and Berger, 2002), such as min., max., median, and other quantiles, provide a better description of the data. For each change size distribution in this section, we first provide its order statistics and visualize its CCDF. The CCDF(x) of a dataset keeps track of the portion of the data at or above x . The CCDF at the minimal value of the dataset is always 1, because the all data points are equal to or greater than the minimal value. As x increases, CCDF(x) decreases. When x is the maximal value of the dataset, there is only 1 data point that is greater than or equal to x , and CCDF takes the value of one over the size of the dataset. The CCDF charts in this dissertation were produced using the ggplot2 package (Wickham, 2009) in R.¹

Following the statistical approach of Clauset et al. (2009), we estimate the parameters of power law distributions that best fit the data, test the goodness of fit, and compare power law with other heavy-tailed distributions to see if any other distribution provides a better fit.

The output of a goodness-of-fit test is a p -value, which quantifies the probability that the dataset is drawn from the hypothesized distribution. The goodness-of-fit test suggested by Clauset et al. (2009) is based on the Kolmogorov-Smirnov (KS) statistic, which measures how well the hypothesized distribution (e.g., the

¹The R source code can be found at:
<https://gist.github.com/linzhp/46e514251a8138ad1656>

best-fit power law distribution) fits the data. Smaller KS statistic indicates better fit. In order to decide whether the dataset is from the hypothesized distribution, we can compare its KS statistic with that of a synthesized dataset of the same size that is drawn from the distribution. For each goodness-of-fit test in this dissertation research, we produce 100 synthesized datasets, and count the portion of synthesized datasets that has KS statistic greater than the empirical data, using that as the p -value. For example, if there are 10 out of 100 synthesized datasets with KS statistic greater than the empirical data, then the p -value is 0.1. If there is none, we report the p -value as <0.01 rather than 0, because if we increase the number of synthesized datasets, it is possible that some of them might have greater KS statistic. As we use 0.05 as the significance level to reject a hypothesis in the dissertation research, there is no need to know the exact p -value when we know it is less than 0.01.

The goodness-of-fit test can reject a distribution with small a p -value, but a large p -value does not assure the dataset is actually drawn from a distribution. In fact, the tails of power law, log-normal, exponential and Poisson distributions are often similar. Goodness-of-fit tests may fail to rule out two or more distributions for the same dataset. One way to compare the competing distributions is the likelihood ratio test. The idea is to calculate the likelihood of each competing distribution given the data. The one with a higher likelihood wins. To compare the likelihoods, we compute the logarithm of the ratio of them, denoted as $\mathcal{R}$, which can be positive or negative depending on which distribution is more likely to produce the data.

However, an $\mathcal{R}$ close to zero may be caused by statistical fluctuation when the two distributions actually have equal likelihoods. We have to decide how far from zero is enough for an $\mathcal{R}$ value to rule out this possibility. Clauset et al. (2009) provides a way to compute the p -value that quantifies the probability we obtain a non-zero $\mathcal{R}$ when the *true* value is zero. A small p -value means that the two distributions should not have equal likelihoods, otherwise it is unlikely to observe such an $\mathcal{R}$.

This research uses an R package called `powerLaw` (Gillespie, 2015) to perform the parameter estimation of heavy-tailed distributions and above-mentioned hypothesis tests.

4.2 Commit Size

Commit size is defined as the number of AST differences across all changed source code files in one commit in the version control system (in our case, Git). This section looks into how changes are distributed among commits, that is, the distribution of commit size across commits.

Not all commits have code changes, because some of them do not modify any Java code, while some others only have code formatting changes and thus are transparent when comparing their ASTs. Commits without any code changes are excluded from this thesis study. Table 4.1 shows very skewed distributions in all projects, where about 90% of commits have less than 200 changes, while a small number of commits have up to 40,551 changes.

Table 4.1: Order statistics for commit size

Project	Min.	Median	90% Quantile	95% Quantile	Max.
jEdit	1	16	164.3	282.15	6960
Eclipse JDT	1	14	124	236	40551
Apache Maven	1	12	106	179	5989
Google Guice	1	26.5	178.9	324.6	3130

The CCDF of commit size distribution is shown in Figure 4.1, in which the x-axis is the number of changes, and the y-axis is the portion of commits with at least that number of changes. On the CCDF charts, the dots in the right side of each plot closely follow a straight line, which is an indication of power law distribution.

To test the power-law hypothesis on each data set, we first estimated the lower-bound x_{min} of the power law region in the distribution, as well as the exponent α of the power law (Table 4.2). With these parameters, we performed Kolmogorov-Smirnov (KS) tests to decide whether it was possible that the right tails were generated by power law distributions. As can be seen from Table 4.2, most hypothesis tests report p -values greater than 0.05 except for Eclipse JDT. Therefore, with a significance level of 0.05, we can only reject power law as a possible distribution of commit size for Eclipse JDT; for others, a power law interpretation cannot be rejected by KS tests.

Comparing Table 4.1 and 4.2, one may find that the power law region only accounts for about 10% of the commits of each project, making it incapable

Figure 4.1: CCDF for commit size distribution.

Table 4.2: Parameters and p -values of fitting power law distributions to commit size distributions

Project	x_{min}	α	p
jEdit	252	2.59	0.55
Eclipse JDT	122	2.11	0.03
Apache Maven	115	2.33	0.32
Google Guice	120	2.42	0.56

of describing the majority of the data. However, the largest 10% commits are the most interesting ones, because they change a very large portion of the software system. Table 4.2 also shows that the exponent α of the power law distributions are all less than 3, suggesting that the variance is infinite, as we have seen in Section 2.1. Infinite variance in the tail of a distribution makes the variance of the whole distribution infinite too. Therefore, if the tails of commit size distributions in jEdit, Apache Maven and Google Guice follow power laws, it is very difficult to estimate their commit size.

Although we cannot reject power law tails in most cases, other distributions might fit the data better. To rule out this possibility, we conducted likelihood ratio tests to compare power law with log-normal, exponential and Poisson distributions, which often have similar tails. To compare with the power law distribution, the x_{min} of each alternative distribution is set to be the same as the lower-bound of the corresponding power law distribution. Then we estimated the parameters of each alternative distribution that best-fits the data down to x_{min} . Finally, a likelihood ratio test was performed to compare the power law distribution with each alternative one. In the output of the tests, a positive $\mathcal{R}$ indicates power law fits the tails better than the alternative, while a negative $\mathcal{R}$ indicates otherwise. The p -value indicates how likely the corresponding $\mathcal{R}$ is produced by random fluctuation.

Table 4.3 shows that power law fits the tails better than exponential and Poisson distributions in all data sets. Although log-normal distribution fits the

Table 4.3: Comparing power law with alternative distributions for goodness of fit to commit size data

Project	pl vs. log-normal		pl vs. exponential		pl vs. Poisson	
	$\mathcal{R}$	p	$\mathcal{R}$	p	$\mathcal{R}$	p
jEdit	-0.251	0.802	3.43	0.000603	1.99	0.0468
Eclipse JDT	-1.62	0.105	5.27	$\ll 0.0001$	2.36	0.0183
Apache Maven	-0.176	0.860	5.79	$\ll 0.0001$	2.57	0.0102
Google Guice	-0.649	0.516	2.09	0.0365	3.75	0.000176

tails slightly better, the difference is insignificant and could be a result of statistical fluctuation.

Note that in the likelihood ratio tests, the lower-bound of the log-normal distribution is set to be that of the best-fit power law distribution in each project. However, the best-fit log-normal distribution may have a different, possibly smaller, lower-bound in each project, since log-normal distributions are at least as good as power law distributions at fitting the commit size greater than the power law lower-bounds. So we further estimated the lower-bound and the parameters of best-fit log-normal distributions, and performed hypothesis tests in a similar way to what we did for power law distributions. Comparing the medians in Table 4.1 and the x_{min} 's in Table 4.4, we can see that log-normal distributions are able to describe a much larger portion of the commits, covering the bodies and the right tails of the commit size distributions.

To sum up, the tails of commit size distributions (the portion above power law lower-bound) can be power law or log-normal distributions, but the body of the distributions (the portion between log-normal and power law lower-bounds)

Table 4.4: Parameters and p -values of fitting log-normal distributions to commit size distributions

Project	x_{min}	mean log	std log	p
jEdit	20	3.4	1.51	0.25
Eclipse JDT	3	2.16	1.76	0.03
Apache Maven	3	2.56	1.66	0.16
Google Guice	17	3.73	1.33	0.61

can be fitted by log-normal, but not power law distribution. Both power law and log-normal distributions are considered *heavy-tailed distributions*,² and their variance are very large, if not infinite. Therefore, regardless of whether the true tail distributions are power law or log-normal, one cannot estimate change size with mean and standard deviation, which is a common approach to estimate a random variable.

4.3 Monthly Change Size

In this section, code changes are binned according to the calendar month in which they are made, and the number of AST differences for each month is used as the measure of monthly change size.

The order statistics of the monthly change size are shown in Table 4.5. Compared to Table 4.1, the variance of monthly change size is less than that of commit size, due to the smoothing effect, i.e., there are often multiple commits in a month. Take jEdit as an example, the maximal monthly change size is only about

²Heavy-tailed distribution:
https://en.wikipedia.org/wiki/Heavy-tailed_distribution

Table 4.5: Order statistics for monthly change size

Project	Min.	Median	80% Quantile	Max.
jEdit	20	1090	2923.2	12593
Eclipse JDT	28	4037.5	10552.2	48126
Apache Maven	2	848	3791.8	13657
Google Guice	1	428	1338	5684

Table 4.6: Parameters and p -values of fitting power law distributions to monthly change size distributions

Project	x_{min}	α	p
jEdit	2765	3.17	0.63
Eclipse JDT Core	12303	3.59	0.91
Apache Maven	794	1.79	0.01
Google Guice	249	1.87	0.14

12 times as large as the median, but the largest commit has 435 times as many changes as the median commit.

The CCDF plots of monthly change size in Figure 4.2 are similar to Figure 4.1, indicating power law tails. The procedure from the previous subsection was performed to test the tail distribution of monthly change size for the four projects and the results are presented in Tables 4.6 and 4.7.

The overall outcome of the hypothesis test is similar to that of commit size

Table 4.7: Comparing power law with alternative distributions for goodness of fit to monthly change size data

Project	pl vs. log-normal		pl vs. exponential		pl vs. Poisson	
	$\mathcal{R}$	p	$\mathcal{R}$	p	$\mathcal{R}$	p
jEdit	-0.794	0.427	1.65	0.0983	2.31	0.0208
Eclipse JDT	-0.0280	0.978	1.974	0.0484	4.79	$\ll 0.0001$
Apache Maven	-2.08	0.0374	0.408	0.683	2.83	0.00442
Google Guice	-1.38	0.167	0.692	0.489	3.50	0.000469

(a) jEdit

(b) Eclipse JDT

(c) Apache Maven

(d) Google Guice

Figure 4.2: CCDF for monthly change size distribution.

distribution with a few exceptions. With regard to the tail distribution of monthly change size for the Apache Maven project, the power law hypothesis was rejected and log-normal distribution fitted the tail significantly better than power law in the follow-up likelihood ratio test. This trend can be visually verified in Figure 4.2c, where the dots form a smooth curve rather than a straight line. In other data sets, both power law and log-normal showed a good fit at the right tails of their monthly change size distributions.

Table 4.6 shows that the lower-bounds of power law regions are high, especially in Eclipse JDT Core project. The variances of the power law regions are very high, as indicated by the small power law exponents α , even if they are not infinite ($\alpha > 3$). Power law regions account for larger portions in monthly change size distributions than in commit size distributions, covering about 20% to over 50% of the months, if we compare the medians and 80% quantiles in Table 4.5 with the x_{min} 's in Table 4.6. Nevertheless, we still want to see whether log-normal distributions can cover even larger portion. Fitting log-normal distributions to monthly change size in the four projects confirmed our conjecture (Table 4.8). Although the variances of log-normal distributions are generally smaller than power law distributions, they are still very large. Even if monthly change size follows log-normal distributions, estimations of monthly change size are still inaccurate.

Table 4.8: Parameters and p -values of fitting log-normal distributions to monthly change size distributions

Project	x_{min}	mean log	std log	p
jEdit	1535	7.84	0.65	0.88
Eclipse JDT	6293	8.78	0.776	0.84
Apache Maven	5	6.66	1.71	0.27
Google Guice	125	6.18	1.18	0.52

4.4 Summary

This chapter first examined the tail distributions of commit size and monthly change size, and found power law to be a plausible distribution in most cases. Combined with findings in earlier studies (Gorshenev and Pis'mak, 2004; Wu et al., 2007; Hattori and Lanza, 2008; Arafat and Riehle, 2009), we can see that the distribution of change size follows power law regardless of the granularity of the changes, be it at AST-node level, line level or file level. This scale-free quality is an important feature of power law distributions. We can also see from Table 4.2 and 4.6 that the exponent α of power law is often less than 3, or even less than 2. According to Section 2.1, this implies that change size cannot be estimated with confidence.

However, we also found that log-normal distributions could be an explanation to the tails of change size distributions too, and they are able to explain a larger portion of change sizes than power law distributions. From a statistical point of view, there is no evidence to show whether power law or log-normal distribution is the *true* distribution of change size. This was also observed by Clauset et al.

(2009), who suggested that it is important to look at their generative mechanisms to make a sensible judgment.

Clearly, there is no incentive for developers to shape the change size distributions into either power law or log-normal forms, both of which have heavy tails. On the contrary, if developers ever had a chance to choose the change size, they would definitely prefer all changes to be small. In fact, software engineering books often characterize a good design as having the property, “with this design, future changes to do X will be small.” What are the underlying forces that are so strong that no developers, collectively or individually, can resist? Let us demystify them in the next chapter.

Chapter 5

Generative Models for Power Law and Complex Networks

The prevalence of power law distributions and complex networks has intrigued scientists across several fields, inspiring them to study the underlying causes. This chapter provides an overview of some of these explanations.

5.1 Generative Models for Power Law

Power law distributions have been found in many areas, such as the magnitude of earthquakes, the size of human settlements, the intensity of wars, etc. (Newman, 2005). The causes of power law distributions have been studied extensively in non-software domains. Mitzenmacher (2004a) first reviewed the generative mechanisms for power law distributions and for the closely related log-normal distributions. He suggested that power law distributions can be produced by both

preferential attachment and optimization, while log-normal distributions can be generated by multiplicative processes. In a multiplicative process, the initial value of a property is X_0 . In each step, the value grows and shrinks by a percentage of the value in the previous step:

$$X_j = F_j X_{j-1}$$

where F_j is a random variable. The logarithm of X_j can be expressed in terms of X_0 :

$$\ln X_j = \ln X_0 + \sum_{k=1}^j \ln F_k$$

As long as the random variable $\ln F_k$ meets appropriate conditions, its sum will converge to a normal distribution according to the Central Limit Theorem (Casella and Berger, 2002), making $\ln X_j$ follow a normal distribution too. Consequently, X_j will follow a log-normal distribution after a large number of steps in a multiplicative process. However, Mitzenmacher (2004a) shows that if the values cannot go arbitrarily close to zero, and instead there is a lower-bound, a multiplicative process would produce a power law distribution instead of a log-normal one. Furthermore, if the number of steps in a multiplicative process is a random variable, following a geometric or exponential distribution, then the result is a distribution with log-normal body and power law tails, which is called a double Pareto distribution.

Mitzenmacher (2004a) also showed that an imaginary monkey typing randomly could also produce power law and log-normal word frequency distributions,

with minor modifications in the model. Mitzenmacher (2004b) proposed a Recursive Forest File model to explain the double Pareto distribution of file size in computer file systems.

Although some models can produce either a log-normal or a power law distribution with minor modifications, one should not have an impression that all generative mechanisms of power law distributions have this property. Newman (2005) focused on power law distributions and reviewed six generative mechanisms, most of which cannot produce log-normal distributions to the best of our knowledge:

1. Combination of exponentials,
2. Inverse of quantities,
3. Random walks,
4. Yule process, another name for preferential attachment,
5. Phase transition and critical phenomena, and,
6. Self-organized criticality (SOC).

Chapter 4 suggested that the commit size distribution could either be a power law or log-normal distribution, so we examined all of the generative mechanisms above, and found preferential attachment and SOC to be the most promising explanations of the emergent phenomena in software evolution. In fact, some existing work shows that preferential attachment is suitable for modeling some properties in software systems (Concas et al., 2006), while the coexistence of power law distribution and long-range correlation indicates the existence of SOC in software evolution (Wu et al., 2007). The rest of this section provides a brief description of

these two generative mechanisms.

5.1.1 Preferential Attachment

Preferential attachment has been used to explain power law distributions in many different areas. The first person to provide this kind of explanation was Yule (1925), who described the size distribution of biological taxa, making it also known as the Yule process.

It was found that the number of species in biological taxa, such as a genus, follows a power law distribution closely. To explain the distribution, Yule's simple model makes the following assumption: species can only be added to genera and never become extinct. A Yule process starts with one genus. At each time-step, a new species is created by splitting an existing species. Then the new species is added to the same genus as the splitting species. Such *speciation* process occurs at the same rate for every existing species. As a result, the more species a genus has, the more speciation events would occur in it, and as a result, the more likely it is to gain more species. The rate of species gain is proportional to the number of species a genus has. The final rule in the Yule process involves new genus creation. Once in every m steps, the newly split species is sufficiently different from the other species in the genus that it becomes first species of a new genus.

When the number of genera is sufficiently large, Newman (2005) shows

that the fraction of genera with k species is given by:

$$p(k) = (1 + \frac{1}{m})B(k, 2 + \frac{1}{m}),$$

where $B(a, b)$ is the beta function. It is also known that when a is large, the beta function can be approximated by a^{-b} , leading to a power law tail. Consequently, the function $p(k)$ has a power law tail too, thus demonstrating that preferential attachment can lead to distributions with power law tails.

The primary limitation of the Yule process lies in its assumption that species do not become extinct, when in fact they do. This is also the limitation of most preferential attachment models proposed to date: they are unable to model the “destructive” process by which entities (e.g., genera) reduce the value of the property (e.g., number of species) in question. To address this shortcoming, Yamasaki et al. (2005) proposed a generalized preferential attachment model that included a process of “preferential detachment”. However, the destructive process itself exhibits complex patterns that requires more sophisticated models, like the one in the next section.

5.1.2 Self-organized Criticality

Disasters are costly, whether they are naturally occurring, such as the mass extinction of dinosaurs and large earthquakes, or occur in human society, such as the Great Depression in the early 1930s. Although these disasters sound very different, there is one shared pattern: they are all part of the same type of

distribution—the power law distribution, which were found in the size of extinction events (Raup, 1986), earthquakes (Gutenberg and Richter, 1955), economical fluctuations (Mandelbrot, 1963), and so on. Bak (1996) argued that these events are caused by self-organized criticality (SOC). During an SOC process, many small changes create cumulative effects, leading the system into its critical point, where avalanches occur. The evolution of such complex systems consists of *equilibria*, where small changes accumulate, and *punctuations*, where large changes, or avalanches, disturb the system. The size of avalanches follows power law distributions (Bak et al., 1988). There is no need to have different models for equilibria and avalanches; they are both outcomes of self-organized criticality.

A classic illustration of SOC is a growing sandpile (Bak et al., 1988). Imagine that sand trickles down at a constant rate onto a flat floor thus forming a pile. Most of the time, individual grains do not move after they land, and new grains only cause nearby grains to move slightly, if at all (equilibrium). As sand grains accumulate, the pile becomes steeper and steeper, eventually leading to a critical point where dropping more grains onto the pile could trigger a sand slide (avalanche), in which many grains are displaced, some of them far away from the newly added grains. If we do not start with a flat floor, but instead start with a sandpile steeper than its slope threshold, then there would be a large sand slide at the beginning, ending up with a sandpile on or right below the threshold. Therefore, no matter where we start, flat floor or a sandpile with any slope, as grains fall onto it, the sandpile seems to be “attracted” to its critical point, where the sandpile barely

maintains its stability. Over the process, different sizes of sand slides occur, following a power law distribution (Bak, 1996).

To better understand SOC, we have to introduce the concept of *phase transition*. Many systems have more than one states. In some circumstances, a system can change from one state into another, for example, from the solid state to the liquid state. In the vicinity of such transition points, or *critical points*, people often observe power law distributions of some measures. The phenomena at the critical point, including power law distributions, are collectively known as *critical phenomena*.¹ In fact, phase transition is one of the six generative mechanisms for power law distribution reviewed by Newman (2005), noted earlier in this section.

The “percolation model” by Newman (2005) can help in understanding phase transitions and critical phenomena. Imaging a large lattice in which squares are colored independently with probability p . A *cluster* is defined as a contiguous region of adjacent colored squares. When p is small, the average cluster size is small. If we increase p a bit, the average cluster size increases too. If p is constant, changing the size of the lattice would not affect the average cluster size. However, when p is large enough, all almost colored squares will be connected together in one large cluster, the *spanning cluster*. This state is called *percolation*. Now the average cluster size is limited by the lattice size: the larger the lattice is, the larger the spanning cluster will become. Changing from a non-percolation state to percolation is a phase transition for the lattice. At the critical point, where $p = 0.5927462\dots$, the

¹Critical phenomena: https://en.wikipedia.org/wiki/Critical_phenomena

cluster size follows a power law distribution.

However, to produce power law distributions with phase transitions, one has to tune the parameters of the system precisely at the critical point, e.g., setting p to the specific value in the percolation model, making it implausible to explain the prevalence of power law distributions in the nature and human society, because it is very unlikely for so many real world systems to coincidentally have the exact parameter values for their critical points. Imagine if we arbitrarily pick a p for the percolation model, how likely is it to get a number in the vicinity of 0.5927462? Nevertheless, Bak et al. (1988) found it possible that some dynamic systems were attracted to their critical points and stay at their critical points indefinitely, regardless what their initial state is. The ability of such systems to organize themselves to their critical points is called self-organized criticality. A system at its critical state exhibits punctuated equilibrium.

Let us continue with the percolation model. Imagine the lattice is a landscape, where a colored square indicates that the space is occupied by a tree. On this landscape, trees grow instantaneously at a constant rate in random unoccupied squares. Once in a while, lightning strikes a random square. If there is a tree in that square, a wildfire starts, burning down the tree and the entire cluster it belongs to (Figure 5.1). After a tree is burnt, the square becomes empty again.

If a process starts with an empty landscape, the forest is sparse, and trees are mostly isolated at the beginning. Lightning, if it happens to hit on an occupied square, can only burn down one or two trees. As the forest grows, larger clusters

Figure 5.1: Wildfire started by lightning that burns an entire cluster (reprinted from Newman, 2005).

of trees form. Eventually, when the density of the forest exceeds a percolation threshold p , all trees in the forest are connected in the spanning cluster. At this point, wildfire starting at any tree will burn down the whole forest, bringing the forest below the percolation threshold. Then the forest grows towards percolation again. In the long term, the density of the forest fluctuates around the percolation threshold. In a large lattice, the fluctuations are small as compared to the size of the system. So the system is approximately sitting at its critical point indefinitely, where the size distribution of clusters follows a power law distribution. Computer simulation reveals that the size of forest fires, as measured by the number of trees burnt, follows a power law distribution. This is the forest fire model, as proposed by Drossel and Schwabl (1992).

One of the earliest models of SOC was that of biological evolution and extinction by Bak and Sneppen (1993). By dividing geological history into stages with the same length, it has been found that the fraction of species that became extinct in one stage as compared to the previous stage follows a power law distribution

(Raup, 1986). Bak and Sneppen (1993) proposed a simple model to explore a possible causal explanation for this behavior. In the model, species are arranged in a circle, each with two neighbors and a random fitness number. At each time step, the species with lowest fitness is eliminated, and replaced by another species with a random fitness. When the simulation starts, the fitness values are uniformly distributed between 0 and 1, so the lowest fitness is often close to 0. As the least fit species is removed and replaced by a new species with random fitness, the fitness of the least fit species is likely to increase. Such increase is not monotonic, as one can imagine the new species happens to have a even lower fitness than the one it replaced. However, if we keep track of the highest fitness of the least fit species we have seen so far, it would increase over time, as shown in Figure 5.2. Eventually, the system reaches a point where the least fitness cannot grow any further. At this point, the fitnesses of all species are on or above a certain self-organized threshold f_c . In the next step, the least fit species, which is right on f_c , is chosen to mutate, together with its two neighbors, starting an “avalanche” of extinction events. The avalanche stops when all the species are on or above the threshold again. Paczuski et al. (1996) monitored the size (number of extinction events) of avalanches in a computer simulation, and found that it follows a power law distribution, similar to that in biological evolution.

While the power law distributions is the most notable feature of an SOC process, there is another important feature, long-range correlation, that is less discussed in the literature. In an SOC process, the size of avalanches naturally

Figure 5.2: The lowest fitness value over time (reprinted from Paczuski et al., 1996). The x-axis is the time, and the dots are the fitness of the least fit species at the time. The step function represents the highest fitness number of the least fit species ever appear before the time.

forms a time series. A widely used analysis for time series is to measure their long-range dependency, which is the tendency for a time series to regress to the mean, or cluster in one direction. The Hurst exponent, H , ranging from 0 to 1, is used to measure such tendency. It was proposed by Hurst (1951) to study the fluctuation of water level in the Nile River. The value of H is interpreted as follows:

- $H > 0.5$: long-term correlated. Large values in the time series are likely to followed by large values, and vice versa. In the case of the Nile River, Hurst found that $H = 0.91$: heavy floods were likely to be followed by heavy floods.
- $H < 0.5$: long-term anti-correlated. The time series quickly alternates between values above and below the mean.
- $H = 0.5$: uncorrelated. Future data points are not affected by the past in the time series.

Table 5.1: Hurst exponent of change size in different projects

	Commit size	Monthly change size
jEdit	0.596	0.334
Eclipse JDT Core	0.608	0.647
Apache Maven	0.591	0.702
Google Guice	0.592	0.363

There are several methods to calculate the Hurst exponent for a time series (Taqqu et al., 1995). Rescaled range analysis is a popular method, which starts by calculating the expectation of rescaled range: $E[R(n)/S(n)]$, where $R(n)$ is the range of the first n values, and $S(n)$ is the standard deviation. Such expectation can be seen as a function of n . By fitting it with a power law function, we get:

$$E\left[\frac{R(n)}{S(n)}\right] = Cn^H$$

where C is a constant, and the exponent H is the estimation of the Hurst exponent.

The avalanches in an SOC process are long-term correlated. Wu et al. (2007) found that the logical and structural change sizes of nine C/C++ software systems were also long-term correlated time series. To verify their finding in change size, we perform a follow-up empirical study based on the same data as in Chapter 3. Table 5.1 shows that commit sizes are long-term correlated in all projects, while the values of H change from project to project for monthly change sizes.

The long-range correlation in commit size indicates the possible existence of an SOC process in software evolution. However, grouping changes by months smooths the time series and diminishes the trend.

5.2 Generative Models for Complex Networks

Some complex networks have power law degree distributions, so the generative models for power laws can be used to produce such complex networks too. Preferential attachment is often used to produce scale-free networks. However, preferential attachment favors older nodes in the network, because older nodes have more time to accumulate connections in the past, and thus are more likely to receive more connections in the future. This is not always true in real world networks. As noted by Bianconi and Barabási (2001), some research papers in a citation network can get many citations in a short time, and some young people in social networks are better at making new friends. As a result, Bianconi and Barabási assigned a *fitness* to each node to indicate their different ability to compete for new links. The probability to acquire new links is then proportional to both the number of existing connections and the fitness of the nodes. Vázquez et al. (2002) further studied this model and found it produced a degree-dependent clustering coefficient.

Realizing that random graphs have a short characteristic path length, but their clustering coefficient is much smaller than many complex networks, Watts et al. (1998) proposed a rewiring lattice model that could produce both short characteristic path length and high clustering coefficient networks, which they call *small worlds*. Instead of a random graph, they start with a lattice, which can be seen as the opposite of a random graph. In a d dimensional lattice, each vertex, except the

border ones, is connected with $2d$ vertices. If one takes such a lattice and connects each vertex with z nearest vertices with $z \gg 2d$, then most immediate neighbors of a vertex are also neighbors of each other, ending up with a highly clustered network. However, the characteristic path length of the lattice is very high if d is small. Watts et al. suggests that if one goes through all the links in the lattice, and rewires, say 1/4 of them to random vertices, then the resulting lattice is a small world.

A model that has strong potential in producing the small-world property in the class graph is the random bipartite graph model proposed by Newman et al. (2002). A bipartite graph has two types of vertices. Edges can only connect two different types of vertices. One example of a bipartite graph is a company director affiliation network, in which the vertices are boards and directors. When a director sits on a board, an edge is established between the two vertices. A bipartite graph can be projected to a simple graph by keeping only one type of vertices and connecting two vertices only if they connect to the same vertex in the bipartite graph. For example, two directors in the unipartite projection are connected if they sit on the same board. Newman et al. shows that bipartite networks with correct degree distributions but random edges have very similar clustering coefficient as compared to affiliation networks in the real world.

A software system can be modeled as a bipartite network with classes being one type of vertices, and methods being the other. The result in Newman et al. (2002) implies that, as long as our simulation model has a correct degree

distribution in the class-method bipartite graph, the class projection will have high degree of clustering, like actual class graphs.

Observing that many complex networks have scale-free topology and a high degree of clustering with degree dependent clustering coefficients, Ravasz and Barabási (2003) proposed a hierarchical network model that is able to produce these properties at the same time.

Chapter 6

Simulating SOC and Preferential Attachment in Software Evolution

From the previous chapter, we can see that preferential attachment is good at explaining the constructive process by which complex systems grow, while SOC provides a plausible explanation for the destructive process by which existing components of complex systems are removed or replaced. The evolution of software systems involves both constructive and destructive processes, both of which account for significant portion of the changes. In this chapter, we construct a software evolution model that uses preferential attachment to simulate the constructive process and SOC to simulate the destructive process. Although preferential attachment and SOC have been used separately to simulate software evolution (e.g., Gorshenev and Pis'mak, 2004; Turnu et al., 2011), having preferential attachment and SOC in one model was never attempted before, to the best of our knowledge.

The novelty poses a research question: can such a model reproduce the punctuated equilibrium in change size observed during software evolution, resulting in a software system exhibiting complex network properties?

The model will be verified by computer simulations. Each simulation run starts with a simple piece of Java source code with one class and one method, which is parsed into an AST. Then a series of simulated changes are made to the AST. The changes include adding classes, adding methods, calling methods from other methods, deleting methods, and adding statements to methods. At the end of the simulation, the resulting ASTs are printed to source code files, thus creating a synthetic software system with random behavior. Hence the intent here is not to synthesize a piece of software that meets some requirement document—instead the goal is to model the low-level processes of software evolution so as to understand where power law distributions and complex networks come from.

This chapter first introduces how the simulation model realizes preferential attachment and SOC when making these code changes. Then the model is presented, followed by the setup of our experiment.

6.1 Preferential Attachment

Preferential attachment is used to simulate the growth of software, including adding dependencies among its components. Although there are several types of component dependencies in software, only method calls and class coupling

are simulated in this model. Based on preferential attachment, the following rules are used during the simulation:

- The likelihood that a method is called is proportional to the number of times it is being called. The assumption is: if a method is used in many places, it indicates the method is more “useful”. In addition, it is more likely to be encountered and known by developers. When developers need a specific functionality, that method is more likely to be used.
- The likelihood of a method or a class to grow is proportional to its size. The assumption is: developers tend to place related code together. When a method or a class is large, it is more likely that some of its code is related to a new piece code in a developer’s mind, and thus the new code is more likely to be added to the method or class.
- The likelihood of a class to be inherited from is proportional to the number of its subclasses, with a similar rationale to method calls.

It has been demonstrated that these simple rules, if running alone, are able to produce power law distributions in class size, number of method calls, and subclasses (Turnu et al., 2011). However, preferential attachment does not model the deletion of entities (in this case, AST nodes) very well, and it does not have a sense of commits or time to measure change size either. To reproduce punctuated equilibrium in software evolution, we need to add additional rules.

6.2 Self-organized Criticality

An SOC process is characterized by its avalanches, such as the sand slides in the sandpile model (Section 5.1.2). Avalanches are analogous to source code changes of different sizes during software evolution, as shown in Chapter 4. To measure change size, we need to group changes in some way, such as by calendar month, or commit, as in Chapter 4. During the simulations, we group changes by artificial commits.

In order to create commits, we borrow a concept from Bak and Sneppen’s model for biological evolution: *fitness*. For a given method in a software class, fitness is a metric of how well it endures throughout a software evolution, with values ranging from near 0 (poor fitness, thus requiring frequent modifications to address issues, and hence very susceptible to change) to 1 (very stable). Initial fitness values are randomly assigned to methods in simulation runs. In each step, the method with the lowest fitness is selected for either a update operation, or for deletion. When a method is changed, it gets a new random fitness value. In addition, when a method interface is changed or deleted, all its callers have to change as well, and thus are assigned new fitness values.

During a simulation run, a threshold, f_0 , is established, and every commit will have no methods with fitness below threshold. Then a commit in our simulated software evolution will include syntactic code changes between a step S_i in our simulation model when all fitness values are above f_0 , and the closest subsequent

step S_j when the minimal fitness value of all methods, $f_{min}(S_j)$ is back above f_0 again. Steps between S_i and S_j constitute an avalanche, with its size being the number of changes made in between.

During an SOC process, the f_{min} will never exceed the self-organized threshold f_c . As a result, setting f_0 above f_c would lead to no commit after the initial fluctuations. When f_0 is set just below f_c , the avalanche (commit) sizes follow a power law distribution (Bak and Sneppen, 1993). f_c can be identified in the simulation result if SOC is produced.

6.3 Simulation Model

The rules for preferential attachment and SOC are implemented as four interweaving processes in the simulation model, corresponding to creating, calling, updating and deleting methods, denoted as P_c , P_r , P_u and P_d , respectively. Each process is associated with a probability p_i , with $\sum_{i=1}^4 p_i = 1$. For each step in the simulation, one of the four processes is chosen to run according to their probabilities.

P_c works as follows:

1. Create an empty method M , and assign it with a random fitness value between 0 and 1;
2. Add M to a class:
 - With probability q_1 , add M to a new class, which is created as fol-

lows:

- (a) Create a empty class C ;
- (b) With probability q_2 , make C a subclass of an existing class in the system, chosen with a probability proportional to the number of its current subclasses;
- With probability $1 - q_1$, add M to an existing class in the system, chosen with a probability proportional to the number of its current methods;

P_r performs the following operations:

1. Choose a method S as the caller with a probability proportional to the size of its body, measured by the number of statements;
2. Choose a method T as the callee with a probability proportional to the number of its current callers;
3. Call T from S , and assign S with a new random fitness number.

P_u inserts a statement into the method with the least fitness value and assigns a new random fitness value to the method. To delete a method with P_d , the following operations are performed:

1. Find the method M with the lowest fitness value;
2. Remove all references to M :
 - If the caller M_c becomes an empty method after removing references to M , add M_c to V ;

- Otherwise update M_c with a new fitness value;
3. Remove M from its class C ;
 4. If C is now empty, delete C and update all its subclasses to inherit directly from `Object`;
 5. Repeat operations 2-4 for all methods in V .

Note that P_u and P_d can lead to ripple effects, causing many methods to be changed or deleted.

6.4 Experiment Setup

In theory, an SOC model does not need fine-tuning the parameters, as the system should be able to organize itself to the critical point. However, in our simulations, we need some control over the parameters, because of two reasons. First, not needing fine-tuning does not mean that we can set the parameters arbitrarily. Take the forest fire model in Section 5.1.2 for instance, if the wildfires were so frequent that it stops any big cluster from being formed, the lattice would not percolate at all, and not SOC would be happening. In our model, if P_d is too frequent, the software system would not grow at all, and it would be hard to observe any power law distributions. This is not the case in the real world either, which is the second reason. Ideally, we would set the parameter values based on empirical data, which can be an extension to this work. For now, p_{1-4} were heuristically set to 0.1, 0.4, 0.45, and 0.05 respectively, while q_1 and q_2 were set to 0.1 and 0.8 re-

Listing 6.1: Initial Java source code

```
public class App
{
    public static void run()
    {
        System.out.println( "Hello_World!" );
    }
}
```

spectively. The simulations start with the source code in Listing 6.1. During each simulation run, 200,000 steps are executed, each executing either P_c , P_r , P_u or P_d . Twenty simulation runs were performed, with results reported in the next section. For each simulation run, we recorded the change size for each step, as well as the resulting source code at the end of the simulation.

For reproducibility, the source code of the simulation is made available on Github.¹

6.5 Simulation Results

This section presents the power law distributions and complex networks that resulted from the simulations. The 20 simulation runs produced slightly different outcomes. In this section, we try to cover as much variety as possible. For scalar measures, the values from all the simulation runs are presented; for distributions, the distribution from a typical simulation run is presented. If there is any simulation run that diverges from the typical one, the most extreme one is also presented.

¹<https://github.com/linzhp/Codevo3/tree/v1.x>

Figure 6.1: Minimal fitness value at different steps. The x-axis represents different steps, y-axis is the minimal fitness value of all methods at the end of each step.

6.5.1 Commit Size

The simulated software evolution exhibited SOC behavior. The scatter plot of $f_{min}(S)$ at different steps during a typical simulation (Run #1) is shown in Figure 6.1. At the beginning of the simulation, f_{min} could be any value between 0 and 1. This is because when there is only one method when a simulation run starts, and f_{min} is the fitness of the method. Over simulated time, the range of f_{min} values reduces as the system grows. The trend is different from that in Figure 5.2 because Bak and Sneppen’s model has a fixed number of species throughout the simulation, but the number of methods in our systems grows over time. Nevertheless, the system became stationary after around 40,000 steps in our simulations, and f_{min} never went beyond a self-organized threshold around 0.5.

With f_0 set to 0.49, we obtained 2489 commits of different sizes. It can be

Figure 6.2: The CCDFs of commit size distributions during the simulations

Table 6.1: Fitting power law to commit size distributions in Run #1 and #3

Run #	x_{min}	α	pl vs. log-normal		pl vs. exponential		pl vs. Poisson	
			$\mathcal{R}$	p	$\mathcal{R}$	p	$\mathcal{R}$	p
1	2	1.53	-1.64	0.1	2.45	0.0145	2.03	0.0428
3	2	1.51	-2.23	0.0255	3.54	0.000551	3.02	0.00255

seen from Figure 6.2a that the CCDF of commit size distribution follows a straight line, indicating a possible power law distribution.

Following the procedure from Chapter 4, we tested the goodness-of-fit of power law to the commit size distributions. Table 6.1 and 6.2 shows that it is risky to rule out a power law distribution, which fits the commit size distribution better than exponential and Poisson distributions. Like in the empirical study, log-normal distributions fits the data slight better, with non-significant difference.

Most simulation runs had very similar commit size distributions, except some rare cases such as Run #3, where the power law hypothesis was rejected (Ta-

Table 6.2: p , H values and graph measures of different simulation runs. The p -values are from the goodness-of-fit tests of power law to commit size distributions (p_1), method fan-in distributions (p_2), class size distributions (p_3), and class collaborators distributions (p_4). p values less than 0.05 are shown in bold. The H values are the Hurst exponents of the commit sizes. The graph measures are: characteristic path length (d) of the largest connected component, clustering coefficient of the entire graph (C) and the random graph of the same size (C_{random})

Run #	p_1	p_2	p_3	p_4	H	d	C	C_{random}
1	0.21	0.81	0.90	0.13	0.584	2.93	0.131	0.0112
2	0.70	<0.01	0.23	0.86	0.635	2.92	0.135	0.0117
3	<0.01	<0.01	0.64	0.66	0.591	2.89	0.139	0.0119
4	0.46	0.06	0.14	<0.01	0.633	2.96	0.154	0.0108
5	0.25	0.18	0.55	0.56	0.539	2.97	0.147	0.0114
6	<0.01	0.79	0.83	<0.01	0.665	2.90	0.121	0.0108
7	0.08	0.10	0.61	0.29	0.636	2.87	0.116	0.0109
8	0.30	0.50	0.80	0.78	0.569	2.87	0.114	0.0111
9	0.55	0.16	0.04	0.05	0.580	2.92	0.144	0.0120
10	0.30	0.61	0.96	<0.01	0.638	2.91	0.124	0.0110
11	0.03	0.77	0.03	0.39	0.620	2.95	0.138	0.0114
12	0.05	0.33	0.77	0.61	0.544	2.89	0.126	0.0121
13	0.30	0.18	0.55	0.24	0.588	2.91	0.133	0.0106
14	0.17	0.90	0.89	0.37	0.555	2.89	0.112	0.0113
15	0.25	0.01	0.07	0.08	0.601	2.89	0.144	0.0123
16	0.55	0.63	0.84	0.44	0.594	2.92	0.140	0.0123
17	0.02	0.03	0.73	0.43	0.632	2.93	0.152	0.0124
18	0.1	0.81	0.72	0.58	0.590	2.88	0.130	0.0115
19	0.21	<0.01	0.24	0.46	0.606	2.93	0.135	0.0119
20	0.41	0.43	0.02	0.36	0.694	2.85	0.120	0.0111

ble 6.2). As we can see in Figure 6.2, the tail of CCDF is more noisy in the commit size distribution of Run #3, and a log-normal distribution fits it better (Table 6.1).

In addition, the time series of commit sizes in all simulations are long-term correlated, as indicated by the H values in Table 6.2. The power law distribution and long-term correlation in commit sizes suggest that our simulation model is self-organized critical.

6.5.2 Distributions of Static Code Measures

As the simulation produced Java source code with valid syntax, static analyses can be performed. Consider simulation Run #1. Its final source code contains 1,518 classes, 9,466 methods and 54,119 lines of code. From the source code, a method call graph and a class graph were built. The method call graph is a directed graph, where each method is a vertex, and a method call forms a directed edge from the caller to the callee. The in-degree of a vertex is the fan-in of a method. The class graph is an undirected graph, in which each vertex is a class, and two vertices have an edge when the two classes collaborate by inheritance or method calls.

It can be seen from Figure 6.3 and 6.4a that the distributions of method fan-in, class size measured by SLOC, and class collaborators all have tails closely following power law, consistent with the empirical studies by Louridas et al. (2008), Herraiz et al. (2011) and Turnu et al. (2011). The results of the hypothesis tests are shown in Table 6.2, 6.3, 6.4 and 6.5. In most simulation runs, power law has a good fit for these distributions. And again, there are small number of cases when the power law hypothesis is rejected and log-normal distributions have a definite advantage.

6.5.3 Other Graph Measures

The class graph of simulation Run #1 has 1,518 vertices and 11,940 edges, with a clustering coefficient of 0.131. The clustering coefficient is comparable to

Figure 6.3: The CCDFs of method fan-in and class size distributions after simulations Run #1

Figure 6.4: Properties of class graph in simulation Run #1

Table 6.3: Fitting power law to in-degree distributions of method call graphs in Run #1 and #3

Run #	x_{min}	α	pl vs. log-normal		pl vs. exponential		pl vs. Poisson	
			$\mathcal{R}$	p	$\mathcal{R}$	p	$\mathcal{R}$	p
1	13	5.07	-0.208	0.835	0.764	0.445	1.83	0.0667
3	7	3.69	-2.57	0.0102	-0.763	0.445	4.26	$\ll 0.0001$

Table 6.4: Fitting power law to class size distributions in Run #1 and #20

Run #	x_{min}	α	pl vs. log-normal		pl vs. exponential		pl vs. Poisson	
			$\mathcal{R}$	p	$\mathcal{R}$	p	$\mathcal{R}$	p
1	69	2.59	-0.464	0.643	0.99	0.0201	3.04	0.00234
20	35	2.24	-1.3	0.193	2.66	0.00772	3.09	0.00202

Table 6.5: Fitting power law to degree distributions of class graphs in Run #1 and #10

Run #	x_{min}	α	pl vs. log-normal		pl vs. exponential		pl vs. Poisson	
			$\mathcal{R}$	p	$\mathcal{R}$	p	$\mathcal{R}$	p
1	37	2.8	-1.02	0.309	1.54	0.124	4.06	$\ll 0.0001$
10	15	2.26	-2.81	0.00489	1.85	0.0646	5.33	$\ll 0.0001$

that in the real projects studied by Valverde and Solé (2003), which is between 0.069 and 0.336. A random graph with the same size would have a clustering coefficient of 0.0112, according to Equation 2.1, which is much less. Figure 6.4b shows that the clustering coefficient is degree dependent, resembling the findings of Myers (2003), shown in Figure 2.1. In the class graph, the largest connected component, in which every vertex is directly or indirectly connected to every other vertex, has 1,459 vertices, and its characteristic path length is 2.93, which is very short. As shown in Table 6.2, other simulation runs have similar values for these measures.

6.6 Summary and Discussion

From the simulation results, it can be seen that the code change mechanisms in Section 6.3, in spite of their simplicity as compared to the real world

software development, are capable of producing the power law distributions and properties of complex networks observed in software systems. As a simulation model that uses random generators, it is possible that some simulation runs go astray. However, the majority of simulation runs gives an affirmative answer to our research questions at the beginning of this chapter.

The change size distribution in Section 6.5.1 exhibits a punctuated equilibrium behavior: a large number of small changes mixed with a small number of large ones. Most changes are localized, and the rest of the system remains stable. New methods often have lower fitness, thus evolve faster and are also more likely to be deleted. However, they tend to be relatively isolated from the other parts of the system, so their changes are more localized. As the system evolves, more method calls are added. Occasionally, when a highly depended-upon method is changed, all its callers are changed too, with new fitness values assigned to them, triggering a ripple effect. Some stable methods are assigned low fitness values and become actively evolving. It takes many steps before all low fit methods evolve to a better fitness or are deleted, and the system is back to equilibrium again. The fact that the commit size distribution can be produced by SOC is an evidence that the distribution is a power law rather than a log-normal one.

The use of preferential attachment in growing the software system during the simulations is generally successful, as it both helps to organize the system into its critical point to allow punctuations, and also produce a component network similar to real software systems. Despite the fact that the model was not explic-

itly configured to produce small world and degree-dependent clustering coefficient, they emerged from the evolution of the class-method bipartite graph.

Although the simulations successfully produced SOC process, the change size distributions it produced is not a perfect match to the empirical data. The commit size distributions in the simulated evolutions followed power law throughout the full range (Figure 6.2), while in reality only the tails follow the power law (Figure 4.1), starting at x_{min} , which is greater than 100 in all four projects studied. Towards the left of x_{min} , commit size follows different distributions, possibly log-normal distributions, like Herraiz et al. (2011) found in file size distributions. It is interesting to know the possible underlying forces of the lower-end deviation.

What is more, this model simulates the collective behavior of all developers when making the code changes. As we are interested in see how low-level changes accumulate to high-level patterns, it is more satisfying to simulate the individual developers directly. Let us bring developers into the simulation model.

Chapter 7

Agent-based Model for Software Evolution

To address the issues in the previous model, this chapter introduces an agent-based model¹ that simulates the interaction between a manager and several developers. Instead of using fitness to define commits, this model uses tasks to group changes into commits. We also give developers some time to understand a piece of software before changing it. The key research questions are:

RQ1. How can preferential attachment be produced in software development? Can power law distributions of method calls and class collaborators emerge without knowing the total number of references to methods and classes?

RQ2. What causes the lower end of commit size distribution deviates from

¹More about agent-based models can be found on Wikipedia: https://en.wikipedia.org/wiki/Agent-based_model

power law?

This chapter first describes the model, and then the simulation results are presented.

7.1 Simulation Model

In the model, there is a manager and several developers, which are all autonomous agents. The manager and developers work following different rules, and communicate through a shared codebase and a task list. Each task in the list specifies a new feature to implement in the codebase. The duration of agent actions is based on a shared internal clock, which has its own time unit independent of wall-clock time.

The manager has two jobs:

- If the task list has less than 10 tasks, she adds a task to the task list every 20 time units;
- Otherwise, she recruits and trains a new developer, which takes 20 time units. After that, the developer starts working on the tasks immediately.

Developers repeatedly take a task from the list and implement the feature specified in the task. Each developer has a memory that keeps track of all methods he has worked on so far. The memory is a list of methods, where duplication is allowed. Every time a developer changes a method, it is added to the front of the list. Therefore, there are multiple occurrences of the same method in a developer's

memory if he has changed the method multiple times.

Before a developer works on a method, he needs some time to understand the method. The more recent the method is in his memory, the less time is needed to understand it. According to Ebbinghaus (1885), memory retention R after a period of time t is:

$$R = e^{-t/S}$$

where S is the relative strength of the memory. The time needed to understand a method is also proportional to its size m , measured by the number of statements it has. As a result, if the most recent occurrence of a method in the memory is at index² i , the understanding time of the method is:

$$T = (1 - e^{-i/S})m + 1$$

In our simulations, S is heuristically set to 40. If the developer did not work on the method before, the understanding time is $m + 1$.

To develop a feature, the developer has to decide the *task size*, i.e., the number of steps needed to complete the task, which is assumed to follow a log-normal distribution in this model. In each step, the developer does the following:

1. Randomly choose a method from the callees of the method changed in the previous step. If this is the first step of a task or the method in the previous step does not call any other method, randomly choose a

²the index in the memory is zero-based

method from the codebase with a probability proportional to its size.

2. Change the method via one of these options:

- Add a statement to the method, or,
- Call another method from the method.

To make a method call, the developer has two options:

- Call an existing method. If the developer “remembers” any methods (i.e., his memory is not empty), a method is randomly chosen from his memory; otherwise, it is randomly chosen from the codebase with a probability proportional to its size.
- Create a new method and call it. The new method will be added to either:
 - An existing class, chosen with a probability proportional to the class size, or,
 - A new class. The new class may inherit directly from `Object` or it may inherit from a class randomly chosen from the developer’s memory.

Since developers “walk” from method to method through method calls, the more times a method is called, the more likely it is to be changed by developers, which is consistent with empirical findings by Vasa et al. (2007). What is more, frequently called methods are likely to have more occurrences in developers’ memories. When a developer needs to call a method, a recurrent method in his memory

is more likely to be chosen. Preferential attachment of method calls is thus implicitly simulated.

When there are no more tasks from the manager, developers refactor the codebase by doing one of the following:

- Rename a method;
- Merge two methods, denoted as A and B below:
 1. Add a parameter to method A
 2. Delete method B
 3. Modify the callers of B to call A instead;
- Move a method to the class where most of its callers are.

After finishing a new feature task or a refactoring, developers commit changes to a virtual repository, and the number of changes committed is counted as the commit size.

7.2 Experiment Setup

The simulation model is implemented with the SimPy library³ in Python. The simulation program is available on Github.⁴ Each simulation starts with the same source code as shown in Listing 6.1. We ran the simulation 20 times, each with 100,000 time units. For each simulation run, the change size for each commit is recorded, as well as the codebase in the end.

³<https://simpy.readthedocs.org>

⁴<https://github.com/linzhp/Codevo3>

7.3 Simulation Results

This section presents the results in the similar fashion as Section 6.5. We choose simulation Run #20 as a typical one to present. For each measure, the result of an exceptional run is also presented.

7.3.1 Commit Size

Figure 7.1a shows the CCDF of the commit size distribution during a typical simulation run (Run #20). It has a straight tail, suggesting a power law distribution. Comparing to the commit size distribution produced in Chapter 6 (Figure 6.2), it also has a curved head, making it more similar to the commit size distribution in real projects (Figure 4.1). The curved head in CCDF is caused by the log-normal task size. Small commits only contain changes with no or little ripple effects. So their size is shaped by the task size, which follows a log-normal distribution. For large commits, ripple effects dominates in the changes, giving it a power law shape.

Statistical tests failed to reject the power law as a possible distribution in the tail (Table 7.1). Compared to other distributions, power law fits the tail better than exponential and Poisson distributions (Table 7.2). While log-normal fits slightly better, the difference is insignificant.

Run #18 produced commit size distributions with non-power law tails, as indicated by p values less than 0.05 in goodness-of-fit tests (Table 7.1). We further

Table 7.1: p , H values and graph measures of different simulation runs. The p -values are from the goodness-of-fit tests of power law to commit size distributions (p_1), method fan-in distributions (p_2), class size distributions (p_3), and class collaborators distributions (p_4). p values less than 0.05 are shown in bold. The graph measures are: characteristic path length (d) of the largest connected component, clustering coefficient of the entire graph (C) and the random graph of the same size (C_{random})

Run #	p_1	p_2	p_3	p_4	developers	d	C	C_{random}
1	0.86	0.74	0.73	0.01	55	2.27	0.112	0.0144
2	0.23	0.8	0.6	0.37	59	2.17	0.0917	0.0116
3	0.26	0.77	0.35	0.67	62	2.13	0.0617	0.0104
4	0.88	<0.01	0.71	0.37	62	2.15	0.0638	0.0101
5	0.4	0.74	0.97	0.74	61	2.14	0.0756	0.0115
6	0.33	0.01	0.03	0.54	67	2.16	0.0632	0.00942
7	0.3	<0.01	0.02	<0.01	64	2.18	0.0827	0.0117
8	0.77	0.27	0.2	0.67	61	2.22	0.0774	0.0110
9	0.73	0.49	0.88	0.92	59	2.09	0.0493	0.00882
10	0.97	0.46	0.07	0.46	62	2.22	0.0721	0.0117
11	0.13	0.13	0.67	0.48	56	2.20	0.0737	0.0109
12	0.69	<0.01	0.3	0.86	55	2.19	0.0882	0.0126
13	0.3	0.26	0.05	0.15	64	2.19	0.102	0.0125
14	0.76	0.01	0.58	0.11	53	2.16	0.0529	0.00956
15	0.62	0.11	0.47	0.17	61	2.23	0.0979	0.0129
16	0.7	0.07	0.14	0.05	52	2.15	0.0686	0.0103
17	0.59	<0.01	0.43	0.8	67	2.19	0.0875	0.0108
18	0.03	0.18	0.39	0.14	57	2.22	0.0997	0.0117
19	0.1	0.63	0.19	0.11	60	2.15	0.0769	0.0109
20	0.44	0.74	0.73	0.58	65	2.14	0.0748	0.0114

Table 7.2: Fitting power law to commit size distributions in Run #20 and #18

Run #	x_{min}	α	pl vs. log-normal		pl vs. exponential		pl vs. Poisson	
			$\mathcal{R}$	p	$\mathcal{R}$	p	$\mathcal{R}$	p
20	24	2.69	-0.425	0.669	3.91	$\ll 0.0001$	3.96	0.000222
18	22	2.53	0.841	0.4	4.54	$\ll 0.0001$	3.36	0.000774

Figure 7.1: The CCDFs of commit size distributions during the simulations

examined Run #18, and found its commit size distribution had a high resemblance to that of Run #20: with similar CCDF (Figure 7.1), x_{min} and α (Table 7.2). Even though the statistical test rejected power law, it is still the best fit among all alternatives.

7.3.2 Developers

At the end of each simulation run, the number of developers does not vary very much, as can be seen in Table 7.1. It ranges evenly from 52 to 67, with both median and mean at around 61. As the system grows, the number of developers needed to evolve the system increases as well (Figure 7.2), although the new features are produced in a constant rate. This is because the larger methods become, the more time it is needed to understand them before making changes. Larger system also allow ripple effects to propagate further, incurring more changes.

Figure 7.2: The number of developers over time in Run #20

7.3.3 Distributions of Static Code Measures

Again, power law behavior is found in method fan-in, class size and class collaborator distributions (Figure 7.3 and 7.4a). The p -values are not significant enough to reject the possibility of power law in most simulation runs (Table 7.1). In a typical simulation such as Run #20, no any other distributions are able to fit the data better than power law distribution (Table 7.3, 7.4 and 7.5). In some simulations, such as Run #7, power law distributions are rejected as the possible distributions of the data, they are still very good approximation, as shown in Figure 7.5.

Figure 7.3: The CCDFs of method fan-in and class size distributions after simulations Run #20

Figure 7.4: Properties of class graph in simulation Run #20

Table 7.3: Fitting power law to in-degree distributions of method call graphs in Run #20 and #7

Run #	x_{min}	α	pl vs. log-normal		pl vs. exponential		pl vs. Poisson	
			$\mathcal{R}$	p	$\mathcal{R}$	p	$\mathcal{R}$	p
20	6	2.48	-0.495	0.621	7.11	$\ll 0.0001$	6.48	$\ll 0.0001$
7	2	2.61	-4.09	$\ll 0.0001$	14.7	$\ll 0.0001$	10.6	$\ll 0.0001$

Table 7.4: Fitting power law to class size distributions in Run #20 and #7

Run #	x_{min}	α	pl vs. log-normal		pl vs. exponential		pl vs. Poisson	
			$\mathcal{R}$	p	$\mathcal{R}$	p	$\mathcal{R}$	p
20	7	1.83	-0.549	0.583	1.52	0.129	1.4	0.161
7	4	1.75	-1.64	0.102	1.55	0.121	1.37	0.169

Table 7.5: Fitting power law to degree distributions of class graphs in Run #20 and #7

Run #	x_{min}	α	pl vs. log-normal		pl vs. exponential		pl vs. Poisson	
			$\mathcal{R}$	p	$\mathcal{R}$	p	$\mathcal{R}$	p
20	7	2.48	-0.495	0.621	7.11	$\ll 0.0001$	6.48	$\ll 0.0001$
7	3	1.99	-2	0.0454	4.26	$\ll 0.0001$	3.27	0.00107

Figure 7.5: The CCDFs of method fan-in, class size and collaborators distributions after simulations Run #7

7.3.4 Other Graph Measures

The simulation Run #20 produced 888 classes, in which 10 classes do not collaborate with any other classes. The other 878 classes form one connected component, which has 4,395 edges. In all class graphs produced by the 20 simulation runs, the characteristic path length is around 2.2 (Table 7.1). The clustering coefficient is close to real projects studied by Valverde and Solé (2003), and is much greater than random graphs. The average clustering coefficient is still negatively correlated to degree (Figure 7.4b). Compared with 6.4b, the negative correlation between clustering coefficient and degree in this simulation model is stronger, especially in low degree vertices. In addition, $C(k)$ in the simulation result of this model and that in the previous chapter seem to follow different functions. However, there is a lack of empirical study to show what $C(k)$ is in real projects.

7.4 Summary and Discussion

This chapter introduced an agent-based model to simulate the interaction between a manager and several developers during software evolution. During the simulation, a commit of changes is made when a task is completed. While working on a task, developers navigate from method to method along the direction of method calls, and remember what methods they have changed. Method calls are made based on individual developers' memory, instead of being preferentially attached to popular methods. As we can see in the simulation results, the power law

distributions of method calls and class collaborators emerge from low-level changes without knowing all their callers or collaborators in the system, giving a positive answer to RQ1.

To address RQ2, we proposed that it might be caused by the task size with a log-normal distribution. In software development, there may be many atomic changes with only one AST node modified, or “one-line changes.” However, it is inefficient to make a commit for each of such changes. Instead, developer may commit them with other changes together. As a result, the commits with only one source code change are not the most frequent ones, contrary to what power law distributions predict. Instead, commits with small number of changes are the most frequent ones, similar to log-normal behavior. We use log-normal task size to model the tendency to group several “one-line changes” in one commit. When a commit consists of changes with little or no ripple effects, the commit size is largely determined by the size of the task that the commit is trying to fulfill. When ripple effects happen, as the method calls graph is scale-free, the number of affected methods follows a power law distribution, shaping the size of the containing commit. Such mechanism is similar to the forest fire model described in Section 5.1.2, as the direct reason for power law fire size in the forest fire model is the power law cluster size at the neighborhood of the critical point.

Chapter 8

Related Models for Software Evolution

On observing power law degree distributions and degree-dependent clustering coefficient in class collaboration graphs and call graphs, Myers (2003) proposed a model to simulate refactoring to produce such graphs. Using binary strings to represent the subroutines and classes, the model simulates three process: breaking excessively long strings into smaller strings, avoiding duplicates by reusing strings with the same content, and avoiding excessive indirection by removing some intermediate strings. Myers found that the properties of software graphs might emergent from refactoring processes like the ones in his model.

Valverde and Solé (2003) modeled class-method association in software systems as a bipartite association graph, and showed that small-world behavior can emerge in its one-mode projections (defined below) when the bipartite associ-

ation is random. In the bipartite association graph $B = (V, U, E)$, $V = \{v_i\}$ is the set of classes, $U = \{m_i\}$ is the set of methods, and $E = \{(v_i, m_i)\}$ is the set of dependencies between classes and methods. It has an edge $(v_i, m_i) \in E$ when class v_i contains method m_i or method m_i uses class v_i . In a bipartite graph, an edge always connects vertices of different kinds. The one-mode projection of a bipartite graph has edges between the same kind of vertices. Each bipartite graph has two one-mode projections: $B_v = (V, E_v)$ (i.e., class projection) and $B_u = (U, E_u)$ (i.e., method projection). In class projection, $(v_i, v_j) \in E_v$ if and only if $\exists m$ such that $(v_i, m) \in E$ and $(v_j, m) \in E$. That is, if two classes are associated with the same method, make an edge between them. Since a method can only belong to one class, the class projection captures "uses" relationships among classes. Method projection is similarly defined. That is, if two methods are associated with the same class, make an edge, thus capturing the union of "uses" and "contains" relationships.

If bipartite association B is random, then the average path length between two classes in B_v is given by

$$D(B_v) \approx \frac{\ln N}{\ln \mu \nu}$$

where $N = |V|$, μ and ν are average method degree and class degree respectively. $D(B_v)$ is normally very small. At the same time, the clustering coefficient is very high, making the class graphs small-world networks.

Gorshenev and Pis'mak (2004) were the first to explain punctuated equilibrium in software evolution with SOC. They adapted the biological evolution model of Bak and Sneppen (1993) to have a random walk to increase or decrease

the system size. There were a one-dimensional and a random neighbor versions of their model. Numerical analyses showed that both versions were able to produce avalanches with power law distributions in their size. They also proved analytically that the random neighbor version was self-organized critical. Another variant of the Bak and Sneppen model was devised by Cook et al. (2005), in which they arranged software components in a ring and assigned a reproductivity to each component to produce new components. Unfortunately, they were not able to produce any power law behavior during their simulation.

Concas et al. (2006, 2007) and Turnu et al. (2011) proposed a modified Yule process to simulate software evolution. They estimated the parameters of the model from empirical data and were able to produce power law distributions in names of instance variables and methods, number of calls to methods, number of subclasses. Zheng et al. (2008) proposed a generalized preferential attachment model, taking into account the aging of nodes and basing the attachment probability on a nonlinear term of node degree. Their simulation showed a degree distribution closely resembled that of the dependency network of Gentoo packages.

Smith et al. (2006) proposed an agent-based model to simulate open source evolution. The model was implemented in NetLogo, in which patches represents software modules that could attract new developers and create new requirements on neighboring patches, and agents are developers working on requirements and existing modules. Their simulation produced similar patterns to the empirical study in terms of complexity, amount of refactoring work and change distribu-

tion. However, they were unable to produce the punctuated equilibrium of system growth.

Stopford and Counsell (2008) developed a framework to simulate the structural evolution of software (i.e., the evolution at the function level). The model separated requirements from software evolution, and modeled each requirement with a *change operator*, which “describes how the requirement will operate on the code base.” By using the default implementation (including default evolution policy, requirement policy and complexity injection policy), they simulated the evolution of software structure, and plotted the changes in several software metrics over time.

Hatton (2009) provided a statistical mechanical model, and mathematically proved that power law distribution of component size is the most likely distribution, given that the system size and the total number of defects is constant, and the number of defects is $d_i \sim n_i \log n_i$ in a component i with size n_i . Using similar approach, but substituting the constraints on the number of defects with a conservation of information, Hatton (2014) modeled software systems as discrete systems and proved that the number of unique tokens in each component would asymptote to power law distribution.

Chapter 9

Future Work

9.1 Extension of the Models

The inclusion of developers in the model introduces new affordances for simulating software evolution. The agent-based model in this research does not consider developers leaving the project, while they do in reality. One could extend the model to simulate such process and observe how that affects software evolution.

Real projects are often constrained by resources, and the development team cannot grow freely as in the model. Setting a cap on the team size may affect the simulation in different ways. The most direct impact is on the productivity. As the system grows, each task takes more time to finish due to the increase of understanding time. If the development team does not grow, the task completion rate will drop. If the manager does not adjust the task creation rate, the tasks will pile up. What is more, if developers are all busy with new feature implementation,

there is no time to perform refactoring. The software system will evolve differently without refactoring.

9.2 Empirical Verification of the Generative Mechanisms

This research proposed models to explain phenomena of software evolution, so one natural extension is to verify whether the generative mechanisms in the models actually exist in the real world.

For preferential attachment, one could examine the revision history, and measure how the value of a property (e.g., the number of callers) at different times in the history affects the later increase of the value. As noted in Section 5.2, preferential attachment gives advantages to old entities to increase their values. For example, an old method may have more time to obtain more callers, which in turn makes it more likely to be called in the future. It is unclear whether such advantages exist in software systems, calling for more empirical studies. If not, some modifications to the preferential attachment in the model similar to Bianconi and Barabási (2001) may be necessary.

The models also assume that large commits are not necessarily caused by any significant external change, such as new feature requirement. We propose that the ripple effect of changes, most likely due to refactorings, can lead to large commits. Hindle et al. (2008) did find that 20.7% of large commits are due to non-functional source code changes, including refactoring. However, their em-

pirical study included non-source code changes as well, and were based on textual comparison, which found that a significant portion of large commits were due to non-source code changes. It is necessary to examine the commits with large number of AST differences, and observe how much the ripple effect in code changes contributes to large commits.

An assumption in the agent-based model in Chapter 7 is that developers navigate from one method to another following method calls when they are making changes, so popular methods (those with more callers) are more likely to be changed. The model also assumes that developers often call methods they have changed before, and developers' productivity of changing a method depends on the recentness of the method in their memory and the size of the method. Empirical studies are needed to verify whether, or to what degree, these assumptions are valid in real projects.

There are many parameters in each models of this dissertation, such as the rate of update, delete operations. They are configured heuristically in our experiments. It is more desirable to configure them according to empirical studies, and explore how their values vary in different projects, and how the variation affects the outcome of the models.

Chapter 10

Discussion and Conclusion

This thesis work starts with a literature review and an empirical study, which reveals several emergent phenomena in software evolution, including the power law distributions and complex networks in software systems. This research is then focused on exploring the generative forces underlying these emergent phenomena.

Using the number of AST differences to measure change size, the empirical study in this thesis research found that the change size in software evolution has a heavy-tailed distribution, either a power law or a log-normal one. Heavy-tailed change size distributions imply that traditional estimation approaches based on mean and standard deviation is incapable of predicting change size. If it is a power law distribution, then the variance is often infinite. This means that the Central Limit Theorem does not apply to such distributions, and hence the sample mean and variance is a poor estimate of the population mean and variance.

Combined with the scale-free nature of power law distributions, there is no “typical monthly change size” or “typical number of changes in a commit”, that we can use for prediction. Even in the best case when it is a log-normal distribution or a power law distribution with finite variance, the variance is still very high, making estimations of change size inaccurate.

The unpredictability of change size has profound implications when we relate change size to development effort or cost. The change effort is likely to be a non-decreasing function of change size. If one considers a commit is a fine-grained unit of accomplishment from development activities, then the cost associated with a unit of accomplishment may be a heavy-tailed distribution too, which makes the effort estimation risky even for fine-grained software development tasks. Looking at a higher level, if the commit size follows a power law distribution, the changes needed for a project may follow a power law too, due to the scale-free property of the distribution. As an important input to the COCOMO II cost estimation model (Boehm et al., 2000), one has to estimate the amount of new, reused and modified code. A power law distribution would make such estimation very inaccurate, if at all possible.

In the empirical study, we also found that large commits tend to follow large commits, while small commits follow small ones. Both power law distributions and long-range correlations are signatures of a SOC process. We then proposed a model to simulate preferential attachment and SOC at the same time, and showed that it could produce the power law distributions and complex networks

found in previous empirical studies. The novelty lies in the use of preferential attachment and SOC in one single model and produce several related phenomena at the same time.

Self-organized criticality in software evolution implies that large, system-wide changes, though infrequent, are inevitable. A significant external (hardware, requirement, market etc.) change is not required to trigger large software changes. As the size and complexity of the system grows, a highly-dependent component will have to evolve at some point, which could trigger large number of changes. One may choose to avoid change such component, but such avoidance could compromise the integrity of the architecture, making the software harder to evolve, and accelerating its decay. To prevent the decay, one has to perform refactoring eventually, which often involves large number of changes too (Hindle et al., 2008). It appears that, as long as a software system evolves, no matter what the developers do and no matter how well-managed is the project, large changes cannot be avoided. In this sense, large changes are Black Swan events (Taleb, 2010). They are rare, but have great managerial and technical impacts on software evolution: they may delay the project schedule, alter software architecture, introduce a new design, etc. Yet, they are unpredictable. In fact, if people know that the software will evolve in a certain way, they would have designed it to make the evolution effortless. In other words, large changes happen precisely because they are not supposed to happen.

To further explore the origin of preferential attachment, and to connect the individual developer contributions to the system-wide phenomena, we propose

an agent-based model that simulate the kinds of changes software developers make on a daily basis as they work on a large project. By having developers navigating along method calls and calling methods they worked before, the model simulates a preferential attachment process and produces a power law structure similar to that in real software system.

We also show that a commit size distribution with a log-normal body and a power law tail can be produced by a combination of log-normal task size and a power law structure: when commits are small, they are shaped by the task size; as the size of commits increase, they are often caused by ripple effects in the changes, and the power law structure shapes the change size. The implication is that power law change size is a result of power law structure, similar to the forest fire model (Section 5.1.2) at its critical point.

This dissertation research not only provides explanations for the emergent phenomena, it also introduces a new approach to study software evolution. Both simulation models in this dissertation research mimic the source code changes in the real world, growing a simple piece of source code into a sizable software system, with syntactically correct and compilable source code. Besides the metrics used in this research, the resulting source code can also be studied with many other metrics, such as the object-oriented metrics suite proposed by Chidamber and Kemerer (1994). One can modify the software evolution models or even develop new models to study how different software development practices may affect the code metrics, which is made possible by the simulation approach in this dissertation.

Bibliography

- Arafat, O. and Riehle, D. (2009). The Commit Size Distribution of Open Source Software. In *42nd Hawaii International Conference on System Sciences*, pages 1–8.
- Bak, P. (1996). *How Nature Works: the science of self-organized criticality*. Copernicus, New York, NY, USA.
- Bak, P. and Sneppen, K. (1993). Punctuated equilibrium and criticality in a simple model of evolution. *Physical Review Letters*, 71(24):4083–4086.
- Bak, P., Tang, C., and Wiesenfeld, K. (1988). Self-organized criticality. *Physical review A*, 38(1):364–374.
- Baxter, G., Freen, M., Noble, J., Rickerby, M., Smith, H., Visser, M., Melton, H., and Tempero, E. (2006). Understanding the shape of Java software. In *Proceedings of the 21st annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications (OOPSLA’06)*, pages 397–412, New York, New York, USA.
- Bhattacharya, P., Iliofotou, M., Neamtiu, I., and Faloutsos, M. (2012). Graph-based analysis and prediction for software evolution. In *34th International Conference on Software Engineering (ICSE 2012)*, pages 419–429.
- Bianconi, G. and Barabási, A.-L. (2001). Competition and multiscaling in evolving networks. *Europhysics Letters (EPL)*, 54(4):436–442.
- Boehm, B. W., Abts, C., Brown, A. W., Chulani, S., Clark, B. K., Horowitz, E., Madachy, R., Reifer, D. J., and Steece, B. (2000). *Software Cost Estimation with COCOMO II*. Prentice Hall, 1st edition.
- Casella, G. and Berger, R. L. (2002). Properties of a Random Sample. In *Statistical Inference*, chapter 5. China Machine Press, Beijing, 2nd edition.
- Chawathe, S. S., Rajaraman, A., Garcia-Molina, H., and Widom, J. (1996). Change detection in hierarchically structured information. *ACM SIGMOD Record*, 25(2):493–504.

- Chidamber, S. and Kemerer, C. (1994). A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493.
- Clauset, A., Shalizi, C. R., and Newman, M. E. J. (2009). Power-Law Distributions in Empirical Data. *SIAM Review*, 51(4):661–703.
- Concas, G., Marchesi, M., Murgia, A., Tonelli, R., and Turnu, I. (2011). On the distribution of bugs in the Eclipse system. *IEEE Transactions on Software Engineering*, 37(6):872–877.
- Concas, G., Marchesi, M., Pinna, S., and Serra, N. (2006). On the suitability of Yule process to stochastically model some properties of object-oriented systems. *Physica A: Statistical Mechanics and its Applications*, 370(2):817–831.
- Concas, G., Marchesi, M., Pinna, S., and Serra, N. (2007). Power-laws in a large object-oriented software system. *IEEE Transactions on Software Engineering*, 33(10):687–708.
- Cook, S., Harrison, R., and Wernick, P. (2005). A simulation model of self-organising evolvability in software systems. In *Proceedings of the 2005 IEEE International Workshop on Software Evolvability*, pages 17–22.
- de Moura, A. P. S., Lai, Y.-C., and Motter, A. E. (2003). Signatures of small-world and scale-free properties in large computer programs. *Physical Review E*, 68(1):017102.
- Deal, B. and Sun, Z. (2006). A Spatially Explicit Urban Simulation Model: Landuse Evolution and Impact Assessment Model (LEAM). *Smart growth and climate change: Regional development, infrastructure and adaptation*, pages 181–203.
- Drossel, B. and Schwabl, F. (1992). Self-Organized critical Forest-Fire Model. *Physical Review Letters*, 69(11):1629–1632.
- Ebbinghaus, H. (1885). *Memory: A Contribution to Experimental Psychology*. York University, Toronto, Canada.
- Fluri, B. and Gall, H. (2006). Classifying Change Types for Qualifying Change Couplings. In *14th IEEE International Conference on Program Comprehension (ICPC’06)*, pages 35–45.
- Fluri, B., Würsch, M., Pinzger, M., and Gall, H. (2007). Change Distilling: Tree Differencing for Fine-Grained Source Code Change Extraction. *IEEE Transactions on Software Engineering*, 33(11):725–743.
- Fowler, M. and Beck, K. (1999). *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Object Technology Series. Addison Wesley, Reading, MA, first edition.

- Gillespie, C. S. (2015). Fitting heavy tailed distributions: the powerLaw package. *Journal of Statistical Software*, 64(2).
- Gorshenev, A. and Pis'mak, Y. (2004). Punctuated equilibrium in software evolution. *Physical Review E*, 70(6):067103.
- Gutenberg, B. and Richter, C. F. (1955). Magnitude and Energy of Earthquakes. *Nature*, 176(4486):795–795.
- Hatton, L. (2009). Power-Law Distributions of Component Size in General Software Systems. *IEEE Transactions on Software Engineering*, 35(4):566–572.
- Hatton, L. (2014). Conservation of Information: Software's Hidden Clockwork? *IEEE Transactions on Software Engineering*, 40(5):450–460.
- Hattori, L. P. and Lanza, M. (2008). On the nature of commits. In *23rd IEEE/ACM International Conference on Automated Software Engineering - Workshops*, pages 63–71.
- Herraiz, I., German, D., and Hassan, A. (2011). On the distribution of source code file sizes. In *International Conference on Software and Data Technologies*, Seville, Spain.
- Hindle, A., German, D. M., and Holt, R. (2008). What do large commits tell us?: a taxonomical study of large commits. In *Proceedings of the 2008 International Working Conference on Mining Software Repositories*, pages 99–108.
- Hurst, H. E. (1951). Long-term storage capacity of reservoirs. *Transactions of American Society of Civil Engineers*, 116:770–799.
- Kohring, G. A. (2009). Complex Dependencies in Large Software Systems. *Advances in Complex Systems*, 12(06):565–581.
- Louridas, P., Spinellis, D., and Vlachos, V. (2008). Power laws in software. *ACM Transactions on Software Engineering and Methodology*, 18(1):1–26.
- Maillart, T., Sornette, D., Spaeth, S., and Von Krogh, G. (2008). Empirical tests of Zipf's law mechanism in open source linux distribution. *Physical Review Letters*, 101(21):1–4.
- Mandelbrot, B. (1963). The Variation of Certain Speculative Prices. *Journal of Business*, 36(4):394–419.
- Milgram, S. (1967). The small world problem. *Psychology Today*, 1(1):61–67.
- Mitzenmacher, M. (2004a). A Brief History of Generative Models for Power Law and Lognormal Distributions. *Internet Mathematics*, 1(2):226–251.

- Mitzenmacher, M. (2004b). Dynamic Models for File Sizes and Double Pareto Distributions. *Internet Mathematics*, 1(3):305–333.
- Myers, C. (2003). Software systems as complex networks: Structure, function, and evolvability of software collaboration graphs. *Physical Review E*, 68(4):046116.
- Newman, M. E. J. (2005). Power laws, Pareto distributions and Zipf’s law. *Contemporary Physics*, 46:323–351.
- Newman, M. E. J., Watts, D. J., and Strogatz, S. H. (2002). Random graph models of social networks. *Proceedings of the National Academy of Sciences*, 99(Supplement 1):2566–2572.
- Paczuski, M., Maslov, S., and Bak, P. (1996). Avalanche dynamics in evolution, growth, and depinning models. *Physical Review E*, 53(1):414–443.
- Potantin, A., Noble, J., Frean, M., and Biddle, R. (2005). Scale-free geometry in OO programs. *Communications of the ACM*, 48(5):99–103.
- Raup, D. (1986). Biological extinction in earth history. *Science*, 231(4745):1528–1533.
- Ravasz, E. and Barabási, A.-L. (2003). Hierarchical organization in complex networks. *Physical review. E, Statistical, nonlinear, and soft matter physics*, 67:026112.
- Smith, N., Capiluppi, A., and Fernández-Ramil, J. (2006). Agent-based simulation of open source evolution. *Software Process Improvement and Practice*, 11(4):423–434.
- Stopford, B. and Counsell, S. (2008). A framework for the simulation of structural software evolution. *ACM Transactions On Modeling And Computer Simulation*, 18(4):1–36.
- Taleb, N. N. (2010). *The Black Swan: The Impact of the Highly Improbable*. Random House, 2nd edition.
- Taqqu, M. S., Teverovsky, V., and Willinger, W. (1995). Estimators for Long-Range Dependence: an Empirical Study. *Fractals*, 03(04):785–798.
- Turnu, I., Concas, G., Marchesi, M., Pinna, S., and Tonelli, R. (2011). A modified Yule process to model the evolution of some object-oriented system properties. *Information Sciences*, 181(4):883–902.
- Valverde, S., Cancho, R. F. i., and Solé, R. V. (2002). Scale-free Networks from Optimal Design. *Europhysics Letters (EPL)*, 60(4):512–517.
- Valverde, S. and Solé, R. V. (2003). Hierarchical Small Worlds in Software Architecture. *arXiv preprint cond-mat/0307278*.

- Valverde, S. and Solé, R. V. (2005). Network motifs in computational graphs: A case study in software architecture. *Physical Review E - Statistical, Nonlinear, and Soft Matter Physics*, 72:1–8.
- Vasa, R., Schneider, J.-G., and Nierstrasz, O. (2007). The Inevitable Stability of Software Change. In *2007 IEEE International Conference on Software Maintenance*, pages 4–13.
- Vázquez, A., Pastor-Satorras, R., and Vespignani, A. (2002). Large-scale topological and dynamical properties of the Internet. *Physical Review E - Statistical, Nonlinear, and Soft Matter Physics*, 65:1–12.
- Šubelj, L. and Bajec, M. (2011). Community structure of complex software systems: Analysis and applications. *Physica A: Statistical Mechanics and its Applications*, 390(16):2968–2975.
- Waddell, P. and Ulfarsson, G. F. (2004). Introduction to urban simulation: design and development of operational models. *Handbook in Transport*, 5:203–236.
- Watts, D. J., Watts, D. J., Strogatz, S. H., and Strogatz, S. H. (1998). Collective dynamics of ‘small-world’ networks. *Nature*, 393(6684):440–2.
- Wheeldon, R. and Counsell, S. (2003). Power law distributions in class relationships. In *Proceedings Third IEEE International Workshop on Source Code Analysis and Manipulation*, pages 45–54.
- Wickham, H. (2009). *ggplot2: Elegant Graphics for Data Analysis*. Use R! Springer, New York, NY, 1 edition.
- Wu, J., Holt, R. C., and Hassan, A. E. (2007). Empirical Evidence for SOC Dynamics in Software Evolution. In *2007 IEEE International Conference on Software Maintenance*, pages 244–254.
- Yamasaki, K., Matia, K., Fu, D., Buldyrev, S. V., Pammolli, F., Riccaboni, M., and Stanley, H. E. (2005). A Generalized Preferential Attachment Model for Complex Systems. *arXiv preprint physics/0502082*, pages 1–11.
- Yule, G. U. (1925). A Mathematical Theory of Evolution, Based on the Conclusions of Dr. J. C. Willis, F.R.S. *Philosophical Transactions of the Royal Society of London. Series B, Containing Papers of a Biological Character*, 213:21–87.
- Zheng, X., Zeng, D., Li, H., and Wang, F. (2008). Analyzing open-source software systems as complex networks. *Physica A: Statistical Mechanics and its Applications*, 387(24):6190–6200.